Final Report

MULTIPROCESSOR ARCHITECTURAL STUDY

By: Alex L. Kosmala, Saul F. Stanten, Woodrow H. Vandever

November 1972

Prepared for the George C. Marshall Space Flight Center,
Huntsville, Alabama 35812, under Contract NAS8-28605

by: Intermetrics, Incorporated
701 Concord Avenue
Cambridge,
Massachusetts 02138

Intermetrics Technical Report #01-73

TABLE OF CONTENTS

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

FOREWORD

This document is the Final Report of a multiprocessor architectural design study, whose objective was to establish a baseline design for a central multiprocessor for a Space Station Data Management System exploiting the NASA/MSFC developed SUMC hardware where possible. The study was sponsored by the NASA Marshall Space Flight Center, Huntsville, Alabama, under contract NAS8-28605, entitled, Research Study on Memory Hierarchy. It was performed by Intermetrics, Inc, Cambridge, Massachusetts, over the period June to October 1972, under the direction of Alex L. Kosmala. Technical monitors for MSFC were Mr. Gerald L. Turner and Mr. James L. Lewis.

Publication of this report does not constitute approval by NASA of the findings or conclusions contained therein.

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

ABSTRACT

This is an architectural design study of a multiprocessor computing system intended to meet functional and performance specifications appropriate to a manned space station application as defined by NASA's Marshall Space Flight Center. Intermetrics previous experience and accumulated knowledge of the multiprocessor field is used to generate a baseline philosophy for the design of a future SUMC* multiprocessor.

The operating system design problem for multiprocessors is to approach the theoretical performance without sacrificing fault tolerance, flexibility, and expandability. Parallel tasking is described as a necessary operating capability in this regard, while exclusive operators are also needed to avoid critical section conflicts. Synchronization, scheduling, and deadlock prevention are other system design features which are discussed, along with memory management. Treatment of the topics of operating system specification and structuring, and the use of a higher order language complete the discussion of multiprocessor operating systems.

Interrupts are defined and the crucial questions of interrupt structure, such as processor selection and response time, are discussed. Memory hierarchy and performance is discussed extensively with particular attention to the design approach which utilizes a cache memory associated with each processor. The ability of an individual processor to approach its theoretical maximum performance is then analyzed in terms of a hit ratio, which is the proportion of time that a memory request can be supplied from cache only. Memory management is envisioned as a virtual memory system implemented either through segmentation or paging.

Addressing is discussed in terms of various register design adopted by current computers and those of advanced design. Using examples, two dimensional addressing, implicit addressing, and the use of descriptors are described. Implementation of a stack-oriented machine is explained, along with the generation of an Effective Address scheme. The overall I/O architecture set forth is upon a Data Bus I/O to service an

---

* Space Ultra-reliable Modular Computer

advanced data bus concept and a Mass Storage I/O. The I/O Controller design is then discussed in terms of interfaces to the processors and to the memories with special emphasis given to recovery from failure.

A complete chapter is devoted to error detection, fault isolation, and recovery philosophy as applied to a multiprocessor system. The important topic of concept verification is given careful scrutiny in terms of

a) analytical techniques and high-level computer simulation, and

b) detailed, low-level simulation.

Finally, the report concludes with a detailed critique of SUMC's architectural characteristics in relationship to the overall design objectives.

Chapter 1

INTRODUCTION

1.1 Scope and Objectives

The work described in this report is the result of a study of multiprocessing system design principles, performed in support of the MSFC in-house multiprocessor computer development. The initial objectives of the study were to achieve a top-level architectural design capable of meeting the functional and performance specifications established for the Phase B Space Station Information Management System Central Processor, and in doing so to exploit as much as possible the current MSFC-developed SUMC processor design. However, during the early phases of the study it became apparent that in order to preserve the value of an independently derived evaluation of multiprocessor design features by Intermetrics, some deviation from these objectives would be necessary. The basic philosophies of multiprocessor design and operation espoused by Intermetrics in defining an architecture appropriate to the Space Station requirements were found to be incompatible with those already adopted by MSFC in arriving at the current SUMC design. Consequently it was mutually agreed that rather than using the existing SUMC design as the basis for the study, Intermetrics should apply the results of their previous experience and accumulated knowledge of the multiprocessor field to establishing a SUMC architecture from an entirely independent point of view. Much of that point of view was gathered in the performance of a previous design study [1] with very similar objectives to those expressed for the SUMC multiprocessor. Although some of the philosophies which are embodied in that design were directly applicable, it was decided not to tailor the complete design to the SUMC application by adopting some features and discarding others. Instead, it was decided to select certain multiprocessor design areas and hardware features and perform an in-depth analysis, review and evaluation for each, in order to establish the philosophies and the rationale developed by Intermetrics in their approach to a multiprocessor design. The objective was to provide a baseline philosophy for the design of a future version of the SUMC multiprocessor, radically different from the one proposed in the present MSFC in-house development program.

In Intermetrics opinion the design of a multiprocessor for a Space Station application should be guided by the following considerations:

a) The performance potentially achievable through the use of multiple processors (often quoted as the main motivation of multiprocessing but, as will be explained in Chapter 2, very difficult to achieve) should not be compromised by implementational incompatibilities, especially in the executive system, nor sacrificed to achieve other MP objectives such as fault tolerance, flexibility, and expandability.

b) Since the overall cost of providing computational capabilities (especially in a difficult environment like a Space Station) may be dominated by software costs rather than hardware, the architecture and operating characteristics of the computer must reflect the needs, desires and techniques of the programmer rather than those of the logic designer.

c) The outstanding advantage of a multiprocessor is its potential tolerance to failures of its components. This capability should be realized in the initial architectural design, and not provided as a final touch after most design decisions have been made.

The detailed analysis of the areas of multiprocessor design which were selected for this study reflect the above basic attitude. They form most of the chapters in the remainder of this report, and include the following topics: Operating System design (Chapter 2); Interrupt Structure (Chapter 3); Memory Hierarchy (Chapter 4); Addressing (Chapter 5); I/O Considerations (Chapter 6); Fault Tolerance (Chapter 7). Additional chapters cover Concept Verification (Chapter 8), since it was of some concern to MSFC how any given multiprocessor design could be given a quantitative evaluation without incurring the initial investment of a hardware build phase, and a critique of the SUMC processor internal architecture (Chapter 9).

Much of the description and terminology found in this report assumes a familiarity with Intermetrics' previous multiprocessor design. To prevent unnecessary (and probably inadequate) repetition of the details of that design, the reader is referred to reference [1]. However, to provide an introduction to at least some of the terms used we present the following overview of the configuration of hardware and software elements of the design, as extracted from sections of reference [1].

1.2 Overview of Intermetrics' Multiprocessor

The basic configuration of the multiprocessor is shown in Figure 1. The MP was specified to consist of a number of

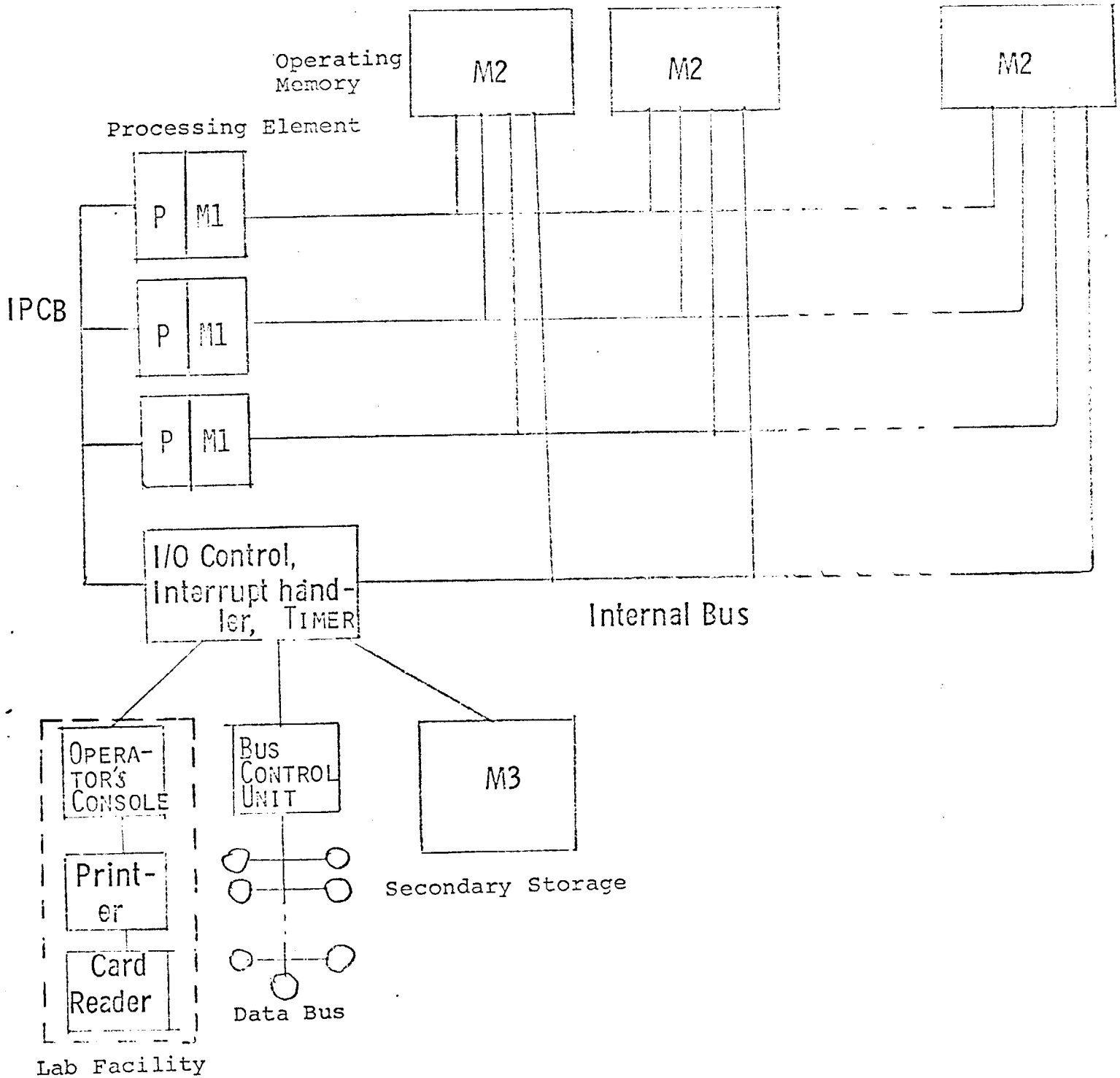


Figure 1: Intermetrics Multiprocessor Basic Configuration

identical, interchangeable processing elements which would execute the major processing workload, and a single, more specialized processor to handle I/O processing and a number of other unique functions. These functions include interrupt handling, interprocessor communication control, and the central timer. The executive was specified to be non-dedicated (to any given processor), and its functions are performed by any of the processors. The choice of which processor is made on the basis of status (e.g., by having completed its current assignment), or by reason of its greatest interruptibility as determined by the priority of its current process. The number of computational processors was specified as three, because the resulting configuration represents the simplest which possesses completely all the characteristics (and problems) of the n-processor case. The two processor system which has received the greatest amount of development and operational experience of all configurations, represents a degenerate form of multiprocessor: while certainly exhibiting true concurrency of processes, nevertheless the dual processor allows certain simplifications of executive functions to be made because of the binary number of active elements in the system. The memory terminology in the figure is used in parts of this report, and is defined as follows:

a) M1: Local memory, dedicated to, and only for use by a processor. This is a general term and refers to all aspects of buffer, scratchpad, control and associative memory, required by a processing element. The contents of any M1 storage cell are available only to the processor of which M1 is an intimate component. Only in case of recovery after a P and/or M1 failure are these contents made available to another processor. In this MP design M1 is not, strictly, a member of the memory hierarchy.

b) M2: Operating memory (main memory, or, in popular terms, "core"). M2 consists of several individual memory modules, all of which are accessible to all processors, including the I/O controller. Each access takes place via a data path dedicated to each processing element, through a port in each M2 module. The basic MP configuration, therefore, requires four ports per M2 module. Each module is fourway interleaved, for purposes of speed, access, conflict resolution, and fault recovery.

c) M3: Secondary storage (backup or Mass Memory). Being a conventional drum or disk, it was decided to interface this level of the memory hierarchy with the rest of the computer system in the more conventional manner, via an I/O channel. The use of

M3 to implement the concept of virtual memory then places the heaviest requirement on the design of the I/O controller and the I/O executive routines.

As mentioned above, several unique functions were gathered together into one, unique module, which is (for convenience) termed the I/O controller (IOC). All interfaces to the outside world were handled via the IOC.

Communication between the processing elements of the MP system (the P's and IOC) were handled by a separate interprocessor bus (IPCB).

(It should be emphasized that the basic configuration of Figure 1 does not indicate the levels of redundancy specified for fault detection and/or recovery. For a discussion of these aspects, refer to Chapter 7.)

The terminology used in this report refers to the way in which information was organized and handled in the previous Intermetrics work. The key terms and their assumed definition are as follows:

a) Program: This is an independently compilable section of code containing pure procedures and/or data.

b) Procedure: A section of code to which execution control can be passed, with or without the passage of parameters.

   1) Internal, not known outside of process (see below)

   2) External, known to name manager and declared in the Process Information Area (see below)

c) Segment: A contiguous block of words defined by a descriptor, which is the unit of memory management.

d) Process: The unit of work as recognized by the operating system. A process is represented by a stack.

e) Stack: Although strictly a LIFO list, the definition of a stack is less rigorous when used to represent a process.

f) Level: A demarcation in the addressing hierarchy. Derived from the concept of lexicographical level in block structured language (such as ALGOL or HAL), but extended to provide convenient addressing by the operating system.

Figure 2 illustrates the relationship and use of some of these terms. Each process is represented by an execution stack. The initial hierarchical level for process execution, and therefore the lowest numerical level for any process stack, is level 2. Subsequent procedure nesting varies the lexical level of each process stack to 3, 4, 5, etc. The portion of a process stack that is below level 2 contains a collection of data termed the Process Information Area (PIA) containing names, priorities, counters, for bookkeeping, etc., specific to each process. Above the PIA the stack behaves more strictly as a LIFO list.

Each process has associated with it a vector of descriptors defining the segments containing the procedures to be executed by the process. These descriptors are addressed as if the vector were a stack: by stack number and offset from the base of the stack. For convenience, this collection of segment descriptors is termed Level 1, since it exists at a more global level than the individual processes, and each such vector will be referred to as a stack (even though, strictly, it is not).

At the most fundamental level there is a single collection of basic system descriptors, variables, etc., which is termed the Level 0 stack, again for convenience of addressing. One descriptor at level 0 points to the stack vector, which contains descriptors of all the stacks in the system including the "pseudo-stacks" of levels 1 and 0.

Each processor contained a set of hardware registers which indicated the actual M2 addresses of the start of each of the system levels, i.e., the base address of the corresponding stack. Figure 2 also shows the linkages that tie the Compool mechanism into the system.

The operating system design philosophy reflected an emphasis on the achievement of reliable operation of both hardware and software. It was assumed that only higher order language(s) would be used in the programming of application software. The exclusive use of HOLs allows secure system operation to be realized without exhaustive runtime verification of each request for OS functions. An intimate and well-defined interface between OS and the compiler(s) was assumed achievable, so that an optimal division between static (pre-run) and dynamic (runtime) diagnosis could be made.

It was assumed that the language/compiler to be used in programming the Space Station application software would possess the facility of handling common data pools (Compools). The MP design provided a Compool implementation.

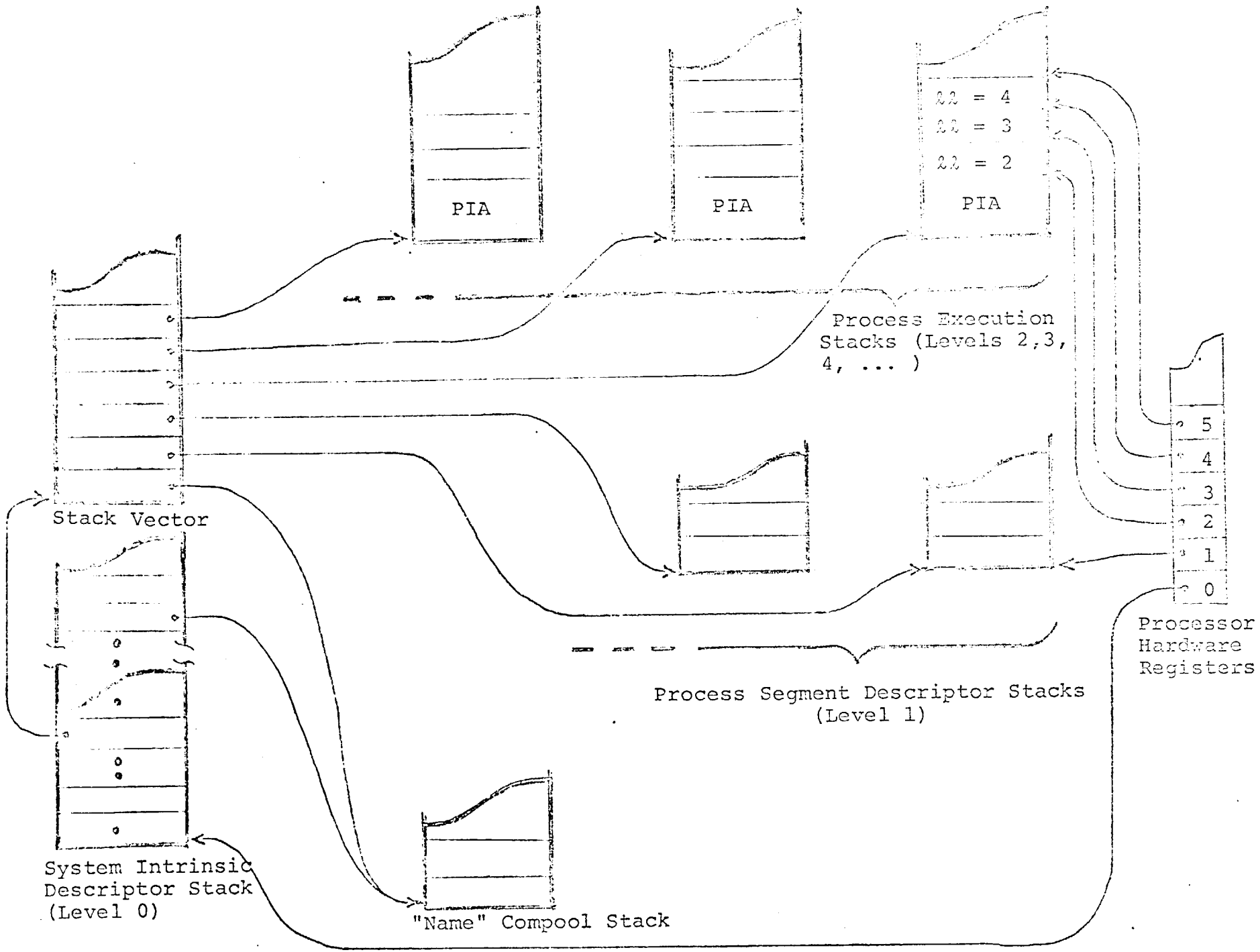


Figure 2: Intermetrics Multiprocessor Operating System Data Structures

Reference for Chapter 1

1. Miller, J.S., et. al., "Engineering Study for the Functional Design of a Multiprocessor Design", Intermetrics/NASA Contract NAS9-11745, September, 1972.

Chapter 2

MULTIPROCESSOR OPERATING SYSTEM DESIGN

2.1 Introduction

This chapter will discuss the special problems facing the designer of an operating system for a multiprocessor computer. The scope of the task which is summarized here did not encompass all aspects of OS design. Emphasis is placed on the more important functions and on those aspects of OS which are unique to, or at least more significant for, multiprocessors as compared with simplex computers.

An operating system for a space station multiprocessor will be capable of supporting a wide variety of functions. Although some of these may be unique to the application, it is very probable that the following standard functions will always be required in some measure:

a) Initialization

This deals with the initial introduction of information into the computing system and its preparation for eventual execution. It includes bootstrapping from a cold start, establishing the minimum state from which the complete system structure can be created, the problems associated with loading and linking of programs and data for execution, etc. This topic is not a trivial one: a real-time MP/OS is a complex structure and the problem of establishing it as a working entity from scratch should be considered at the time its initial design is undertaken. Initialization will not be discussed further.

b) Process State Controller

The basic element of computational work will be termed a Process. Processes can exist in various states: execution, readiness, stall or suspension. This function of the OS controls the orderly progression of processes between these states in response to various stimuli, such as voluntary process state changes, I/O interrupts, priority changes, interprocess communication, etc.

c) Interrupt Servicing

A real time, general purpose, central computer for a space station will almost certainly be required to handle system-originated external interrupts in addition to interruptions due to arithmetic traps and other error conditions. This OS function implements the desired responses to randomly occurring events of this nature.

d) Timing and Synchronization

This function provides the basic mechanism for controlling the time dependent execution, and the synchronization of parallel, concurrent processes in a real-time, multiprogrammed environment.

e) Resource Management

This is the basic function of an operating system. The resources required by a computational process are various. First, there are the basic hardware elements: the processors, memory modules, and interconnecting data paths which must be available to allow the process to run. Then there are the less tangible items such as common programs and data over which conflict of access by several concurrent processes is possible. Lastly, there is external device availability: sensors, avionics data buses, disks, tapes, etc. The resource management function is usually divided into processor allocation, memory allocation, compool and shared data management, I/O and file management. It is the function of resource allocation to ensure that each scheduled process is granted a sufficient share of the available resources to execute in a timely fashion without adverse effect on other processes.

f) Configuration Control

In a fault tolerant computer, the current status and the configuration of all elements of the computer must be continuously monitored and controlled by the operating system.

g) Operator and User Interfaces

The OS must provide facilities to interface with the operator and/or user. For a complex system this is not a simple task, especially when a major mode of operation is interactive usage, by the crew members in controlling the progress of a mission.

h) Performance Monitoring

This is an often under-emphasized function of an operating system, but it is an especially important one in a new or novel application such as a space station MP. The more sophisticated a system is the greater is the need to measure, evaluate and influence its performance.

Some of these functions will be reviewed again in the light of the following discussion of problems facing the multiprocessing operating system designer.

2.2 Problems of Multiprocessing

The multiprocessing environment does not pose any difficulties that the designer of an operating system for a multiprogrammed, single processor system has not also had to face and overcome. The MP adds new facets to familiar problems, however, by reason of the concurrent, rather than sequential, execution of the multiple processes within the system. This requires that greater care be taken to prevent damaging interaction between processes at a point of commonality, especially with regard to shared data. Measures taken to protect processes against each other usually affect performance unfavorably. The maintenance of performance near the theoretical limit is, in any case, more difficult for a multiprocessor than for an equivalent simplex computer.

An attractive feature of the multiprocessor is the prospect of increased performance achieved by means other than advances in processor technology, i.e., n similar processors doing the work of one n times as fast. In practice several factors prevent this promise from being fulfilled. If we define "throughput" as the integral over time of the rate of "useful" computation C, then it can be shown that:

$$\int_0^T C\,dt \geq \int_0^T n\left(\frac{C}{n}\right) dt$$

where n is the number of processors. C is a discontinuous function of time, and as n increases, it becomes increasingly difficult for C to remain non-zero for long periods of time. Computation lost whenever C falls to zero may not be made up in time, and the right hand (multiprocessor) integral continuously loses ground to the left hand (simplex computer) integral. The reasons for this are enumerated below.

2.2.1 Parallelism

In order for all n processors to be kept usefully at work, their load must be capable of being organized into n or more tasks which can be executed in parallel, continuously and simultaneously. The degree to which this can be done depends on the parallelism inherent in the work load. Certain types of computation exhibit natural parallelism, e.g., signal processing, where the same operation is applied to multiple sets of input data (promoting the design of so-called Single Instruction Multiple Data (SIMD) computers, for example the Goodyear Associative Processor [1]). But, in general, parallelism must be sought out, identified and utilized. It exists potentially on several levels:

a) On the "job" level. In a general purpose computer facility, the submitted jobs are normally completely independent of one another, even if they share resources.

b) Within a job, at the task level.

c) Within a task, most of the statements are independent of one another.

d) Within a single statement some computations can be done in parallel.

Parallelism of types c) and d) is not visible to the operating system, because the basic unit of OS is the process (or task). For the type of application being considered for the SUMC multiprocessor, it is not likely that the work load will totally resemble that of a ground based general purpose facility, although it will exhibit more of its aspects than will a simple flight control computer. Parallelism of type a) will probably not be present in sufficient proportion to provide the sole guarantee of full employment for two or more processors. It becomes necessary to deal, additionally, with parallelism at the task level. The trouble is that problem solving with a computer is, in general, a serial process: programmers do not naturally think in terms of concurrent parallel processes in arriving at their solutions, unless such a structure is inherent in the problem. A real time control function may conists of several, more or less independent, activities going on in parallel, e.g., system monitoring, navigation, display processing, and vehicle control. Even so, it is anticipated that there will not be sufficient functions of this type to keep two or more processors fully occupied, all the time.

It is necessary, therefore, to uncover task parallelism that may not be apparent, and even to create parallelism if none

exists. This imposes a constraint on the programmer, which must be considered deleterious because it is not natural. So it is necessary to assist the programmer with a programming language and a compatible operating system that contain features, attractive to use, that encourage the creation of multiple, independent processes. The use of a block-structured language encourages programs to be written as collections of small, closed subroutines. ALGOL, PL/I and HAL are among the languages that possess this property. In addition to structure, a language can provide a convenient and natural way to interface with the executive by recognizing tasks as syntactical entities. The multi-tasking features of PL/I and HAL encourage the programmer to think as he programs in terms of processes which are amenable to scheduling.

The multiprocessor operating system must support the requirements of parallel tasking by providing adequate communication and synchronization primitives, and by protecting shared data against conflicting concurrent accesses. These requirements are discussed in more detail later.

2.2.2 Exclusive Sections

In a general purpose multiprocessor certain operations are concerned with the manipulation of unique system data such as, for example, information maintained by the Process State Controller, which contains the current dynamic state of all processes. Execution of the Process State Controller is an exclusive operation: only one process may perform it at a time. In a simplex computer this is achieved trivially: it is only necessary to inhibit interruption of the single processor by external happenings to assure exclusive execution of the Process State Controller. A multiprocessor requires a more elaborate mechanism to prevent the simultaneous execution of such critical functions by two or more processors. Such mechanisms cause the conflicting processes to become serialized in time, each being admitted to the critical section through interlocking turn-stiles (a generalized mechanism is described later). The net effect is that whenever two or more processes wish to enter an exclusive section, only one may do so and continue executing: the other(s) must wait. If the exclusive section is designed to inhibit the alternate assignment of the processor (e.g., if it is the Process State Controller), then throughput temporarily falls until the other processor is through with the exclusive section. This loss of throughput cannot be made up again. Note that in a batch environment conflicts of this type are rare, but in a real time system of short tasks, with frequent process state changes, the probability of conflict may become significant. This precipitates the following quandary: to encourage parallelism a multiprocessor program should consist of many concurrent tasks, but to

avoid critical section conflict it should be organized into as large a serially-executable piece as possible!

2.2.3 Shared Data

There is a problem with shared data, aside from the need to protect it from simultaneous modification. It is associated with the creation of copies of shared data. In many computer designs, performance improvements have been achieved by localizing lengthy sequences of operations within the fast logic of the processor, rather than executing out of main memory. (The cache memories of the IBM 370 series [2] and the task memory of the Navy's AADC [All Applications Digital Computer][3] are examples of localized processing.) The problem arises because data is maintained local to the processor. If the data is shared with other processes, changes in the original or any of the copies must be reflected in all. Some means must be found either

a) to allow one process access to another's local storage,

b) to update all copies of shared data at the same time or

c) to prevent old values from being used by other processes until updating is performed.

It should be pointed out that this phenomenon is encountered whenever copies of shared data are created in any system: in the Burroughs B6700 series the problem arises through its use of descriptors. These are maintained in the stacks of individual processes. Whenever a descriptor needs to be changed (it is a common occurrence in a virtual memory system for a descriptored item to be transferred to back-up storage: the address field of its descriptor most be modified to reflect this change of whereabouts), all processors in the B6700 are stopped, and all process stacks in main memory are searched for copies of the particular descriptor. The B6700 was not designed as a real time controller so the ensuing loss of processing time was not considered objectionable by the designers. It is a different matter for a space station computer, however. The Multiprocessor design developed by Intermetrics [4] employs a unique approach to a similar problem. The copy of a descriptor may be maintained in an associative memory local to a processor. This avoids accessing the descriptor through three levels of indirection each involving main memory references. Changes in the descriptor are very quickly signalled by the provision of a specific machine instruction which cancels the appropriate entry in the associative memory. The Intermetrics multiprocessor avoids local copies of the data itself, and thereby foregoes the potential performance advantages of local buffer or cache-type processing.

2.2.4 Conflict Over System Resources

The most critical resource is main memory. As the number of processors increases, the possibility of conflict between them over the use of memory increases. As in the case of shared data, a resolution of conflict results in one or more processors losing processing time, and the right hand integral of the expression for throughput given earlier again loses to the left hand. The device of interleaving the modules of a memory system can be used to minimize the delays incurred by conflict, but it exacts a cost in added hardware complexity. Its effect is to randomize memory usage and thus to obtain stationary behavior. Another approach is to partition memory among the various processes so that processors tend to execute out of physically separate modules. i.e., make the memory usage very deterministic. This technique implies a sophistication of the operating system, a well-known job stream, and a memory system of sufficient modularity.

The network interconnecting processors, memories and I/O units is a more critical element in a multiprocessor than in a simplex system. With more than one processor requesting memory at a time, this bus itself becomes a source of conflict. It would seem that a technique that lowers the frequency of use of the bus would lessen the probability of such conflict. For example, the use of a cache memory, by encouraging local execution, would appear to make bus use less frequent. However, analysis shows that the probability of bus conflict actually increases with increasing speed of the cache, thereby defeating any performance advantage.

In summary, techniques devised to minimize conflict in a multiprocessor are susceptible to the following drawbacks, any or all of which combine to prevent the multiprocessor throughput from equalling that of the equivalent simplex processor:

a) Increased hardware complexity and cost,

b) Increasing operating system sophistication, usually accompanied by increased overhead in space and time.

c) Reduced throughput due to delays introduced to resolve conflict.

The more processors in the system, the more marked is this effect. Only in a particular application, for which the characteristics of the work load can be anticipated, is it possible to deduce the number of processors required to achieve a given performance cost effectively. In the absence of such information about the environment of the multiprocessor, this

limit is very difficult to determine. As a result, almost all practical designs of multiprocessors to date have been limited to the degenerate case of two processors. Some designs have even dedicated functions or resources to each processor in order to avoid some of the above problems, resulting in configurations of dual computers rather than dual processors.

2.2.5 Overhead

The preceeding sections have cited several factors that contribute to the complexity of functions that a multiprocessor operating system is required to perform. Each factor contributes to the overhead of computational time and memory space consumed by the operating system. Matters are further aggravated because the many activities going on simultaneously in a multiprocessing environment take on the characteristics of a queueing problem: their deleterious effects are in general worse than additive, i.e., the loss in real throughput is a non-linear function of the number of contributing overhead mechanisms.

But to end this section on a positive note, it should be realized that this depressing parade of multiprocessing difficulties has a corollary: small efforts to limit the damaging effects of each of the mechanisms discussed in this section can yield dramatic improvements in throughput because of the exponential nature of their interaction.

2.3 Exclusion and Synchronization

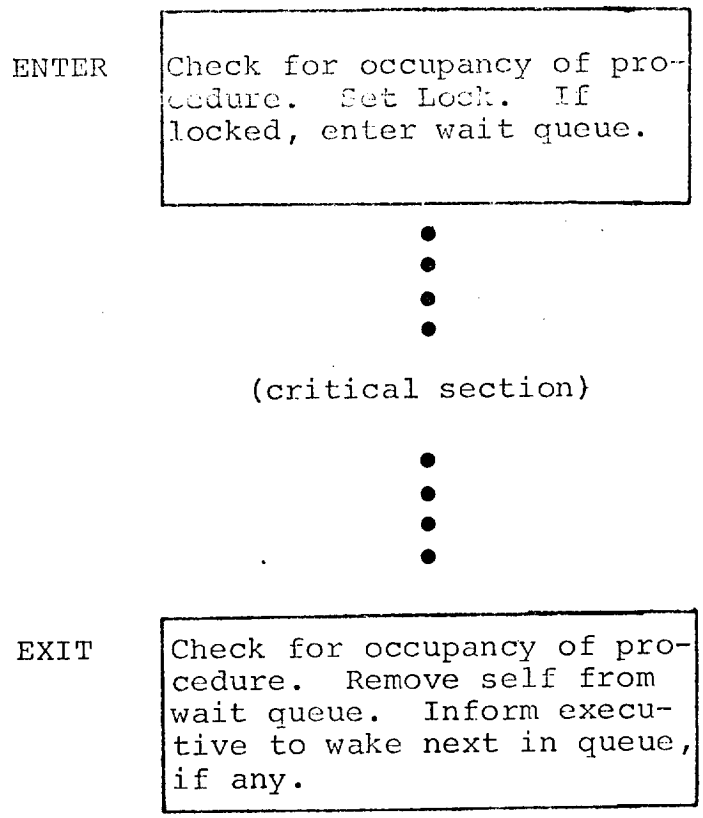
Any multiprogrammed system requires operating system primitives for the communication and mutual protection of the concurrent processes. In a multiprocessor, these activities can be actually time-concurrent and these primitives must be implemented in a combination of hardware and software. The problem of protection against unwanted interactions will be reviewed first, followed by a discussion of synchronization.

2.3.1 Exclusion Primitives

In a simplex computer a basic exclusive operation may be implemented in software, but a multiprocessor needs hardware assistance for such an operation, because of the true time-concurrency of execution of two or more processes. The hardware must be capable of reading the value of a variable, and then rewriting the variable with a new value in one uninterruptible operation. An example of such an instruction is the TS (Test and Set) of the IBM 360 series, which writes all ones into a specified byte and sets a condition code with the original

contents. The Burroughs B6700 RDLK (Read with Lock) instruction, which stores the contents of the B register into the location whose address is contained in the A register, but leaves the previous contents of the location in the B register, is closer to a generalized non-divisible read and write operation.

The actions of a set of general operating system procedures designed to provide the exclusion primitive are as follows:



How these actions are implemented using a fictitious non-divisible read and write instruction NDRW is illustrated in Figure 1. Let the execution of NDRW exchange the contents of the operand, MUEX, with the contents of the accumulator. MUEX may contain the following values:

0	No process executing critical section (i.e., section is "free")
1	Critical section is being executed by one process

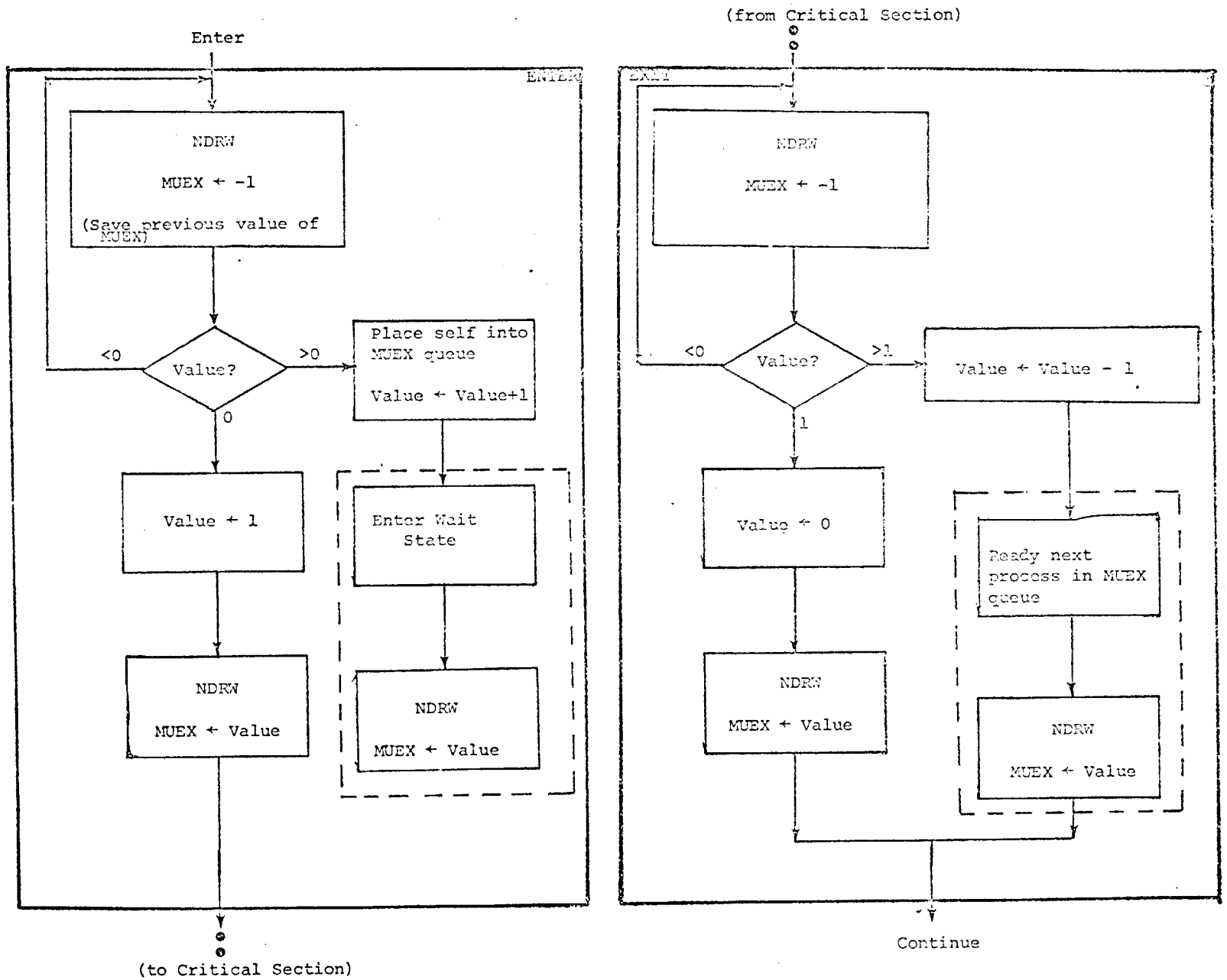


Figure 1: Mutual Exclusion Primitive

2,3,...n	Critical section is being executed by one process, and 1,2,...n-1 are waiting to gain access. Requires a MUEX queue structure to be maintained by OS.
negative	Procedures ENTER or EXIT are being executed by a process. (The OS primitive itself must be protected against multiple use.)

The actions surrounded by dotted lines indicate the execution of the Process State Controller function. Note that the final updating of MUEX in cases where a process is to be placed in the wait state, or readied to execute the critical section, must be done within the Process State Controller to prevent interruption of the sequence.

This exclusion mechanism must be expanded if it is required to accomodate the comprehensive Update Block capability, for controlling the accessing of common data, provided in the HAL language [5]. It is not always necessary to prevent all types of access to shared variables: a shared variable can be read, as long as it is not actually being changed. The ability to differentiate between types of access reduces the time for which a requesting process must be made to wait, with consequent improvement in throughput. The HAL Update Block is in effect a modified form of critical section. Every variable that is addressed within an Update Block has associated with it a "lock-type" attribute. The lock can assume the following states:

a)	Free:	Unlocked
b)	Read:	Accessed for reading only
c)	Copy:	Accessed for modification
d)	Write:	Being modified

A variable that is to be modified is first copied, and all intermediate computations are performed on the copy. This is the meaning of the "Copy" state. Final values are written from the copy to the actual variable after the state of the lock has been raised to "Write". The testing and setting of the states of locked variables requires the use of the NDRW instruction. A requesting process is allowed into the Update Block only if the type of access requested is compatible with the current state of all locks within the block. For example, a request to read the variables is allowed if the current state of all locks is "Free", "Read", or "Copy", but is not allowed if any are in "Write".

An operating system mechanism to implement Update Blocks involves the maintenance of linked queues (see Figure 2). Every locked variable has associated with it a queue of requesting processes, each identified with its individual access type. All queue elements associated with a given process are also linked, to facilitate the response to changes of state of the processes.

The Intermetrics design of a multiprocessing operating system [4], defined a pair of generalized primitives, ACQUIRE and RELEASE, of the form: ACQUIRE (Mode, Category, Name, Access) where each of the terms has the following meaning:

a) Mode: The calling process is placed in the Wait state if access is not immediately possible, or an immediate return may be specified with an indication of why access could not be allowed.

b) Category: Data, code or device. The ACQUIRE primitive is applicable to the protection of shared data, the implementaiton of exclusive sections, or the use of a shared device such as a printer.

c) Name: Identifies the item in the category, e.g., the name(s) of the specified shared variables.

d) Access: Shared, update or exclusive access request. These are analogous to HAL's Read, Copy, and Write lock type states.

It is possible to define any type of required exclusive operation in a given system with these two primitives.

2.3.2 Synchronization

In order to provide for communication between parallel processes of a multi-tasked environment it is convenient to invoke the concept of an "event". An event is a variable whose state reflects the occurrence of an activity within the system, e.g., the completion of a lengthy computation or the arrival in memory of a previously requested item of I/O. The process awaiting the activity is associated with the event. The "signalling" of the event results in the process being made ready to continue. For illustration, let Tasks A, B and C be three independent tasks, all scheduled during the execution of some master Program. Suppose it is appropriate to schedule Task C only when certain computations have been completed by Tasks A and B. Tasks A and B may be executing on separate processors, and thus be unaware of

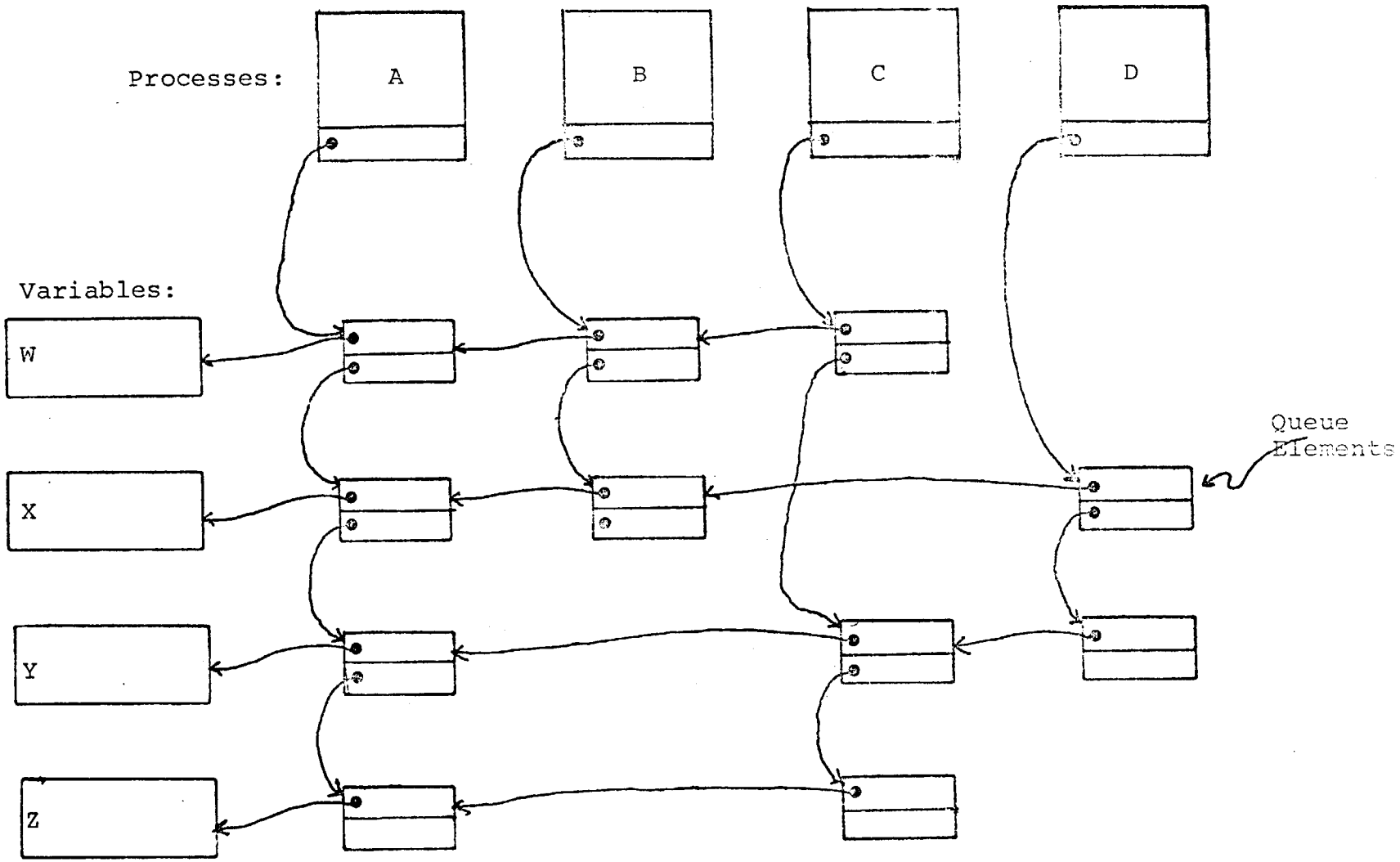


Figure 2: Update Block Structure

one another. In which case, they cannot easily cooperate in the scheduling of Task C. However, if each were to signal an event on completion, e.g., EVENT_A and EVENT_B respectively, then the event mechanism can provide the synchronization that causes Task C to be scheduled as soon as both EVENT_A and EVENT_B have been signalled.

The language multi-tasking features that were advocated earlier to help keep a multiprocessor busy are supported in PL/I, ALGOL and HAL by event mechanisms of varying sophistication. The Intermetrics multiprocessor design [4] specified a very comprehensive event structure which enabled complex logical expressions to be evaluated as event signals. In this design events are controlled by primitives of the form

$$\left\{\begin{matrix}\text{SET}\\ \text{RESET}\end{matrix}\right\} \quad (E,\ n,\ E_1,\ E_2,\ \ldots,\ E_m)$$

which is interpreted as "set (reset) event E when n of the events in the list E_1 through E_m, are signalled." If n = m, this expression is the boolean "and" of all listed events, and if n = 1 it is the "or". The primitives also have a simpler form

$$\left\{\begin{matrix}\text{SET}\\ \text{RESET}\end{matrix}\right\} \quad (E)$$

Response to the signalling of events is basically of two forms:

$$\text{WAIT}(n,\ E_1,\ E_2,\ \ldots,\ E_m)$$

and

$$\text{ON}(n,\ E_1,\ E_2,\ \ldots,\ E_m)\langle\text{code}\rangle$$

In the first, as the WAIT is executed the process is placed in the Wait state until the event expression becomes true. The second statement causes an interruption of the process as soon as the expression becomes true, to execute the procedure specified in the "code".

The implementation of an event structure involves multiply-linked queues of event elements which allow the associations between the processes involved in declaring, signalling and responding to events to be established, executed, and removed in a dynamic fashion. It is perhaps superfluous to point out that such a mechanism in a multiprocessor environment requires processors to be able to interrupt one another. This ability is provided, for example, in the Burroughs B6700 by the "HEYU", and in the RCA 215 by the "INTERRUPT CPU" instructions.

2.4 Scheduling

The scheduling function of the operating system ensures that processes are prepared for timely execution with due regard to their relative importance. It involves some of the functions of Process State Control and Resource Allocation defined earlier. This section will discuss briefly the following aspects of this function:

a) Ensuring that computation time and space are properly apportioned among the processes according to predetermined needs, while maintaining an optimal balance between the conflicting requirements of throughput, efficiency, and response. Throughput is defined as the amount of useful work accomplished by the total multiprocessor system, efficiency is the degree of utilization of the basic components of the system (e.g., processors, memory modules, I/O devices), and response is the ability to react to a given stimulus.

b) Ensuring that competition between processes in their demands for resources do not produce catastrophic conditions, such as deadlock or thrashing.

c) Preventing the resulting computational overhead, especially of time in a real-time control system, but also of space, from becoming excessive (the definition of "excessive" is not attempted here!).

2.4.1 Space and Time Allocation

The computational activities in a space station multiprocessor are expected to fall into the following categories:

Category	Characteristic Response Range	Time Criticality	Examples
Batch	10 secs-mins.	non-critical	Lengthy computations. Off line experiment data processing
Interactive	0.1 sec-10 secs.	non-critical	Crew operational sequences. Time sharing by scientific personnel.
Real Time	1 ms-100 ms	non-critical	Control of scientific experiments. Operational equipment status monitoring
Real Time	1 ms-100 ms	critical	Operational equipment servicing: strapdown IMU. Closed loop control: autopilots, etc.

Processing tasks in the batch category can, to an extent, ignore the constraint of time. The allocation of memory space or other system resources such as common data, input file, I/O devices, processors, can be considered with more freedom. The presence of this category in the total work load can provide a measure of global optimization in the use of system resources to maximize efficiency.

The time-critical real-time tasks can not make such compromises. Resources must be ready when needed. The need is often (but not always) randomly determined. Unless it is composed of highly repetitive tasks, the real-time component of the work load prevents high values of throughput and efficiency from being attained.

A work load consisting of components from each category must be so arranged and presented to the computer system that all tasks can get sufficient cuts at the system's processing resources. Obviously, no amount of intelligence built into an operating system will supply enough computational resources to a work load whose demands exceed the capability of the machine. An operating system can be designed to contain features and to operate in a way that matches the characteristics of the work load. But it remains the responsibility of the user of the system to assign a given work load to the machine in such a way that it does not overload the system.

Task scheduling can be approached from two extremes:

a) Synchronous, or time slot scheduling. Each task is allotted a different, but fixed, interval of time for

execution, which is available at multiples of fixed minor cycle intervals.

b) Demand Scheduling. Tasks are allocated processors and other resources on demand, at execution time, according to the needs and importance of the task and the availability of the resources. Tasks are differentiated in importance by a priority value which stays as initially assigned, or changes as a function of time or the tasks' status.

The advantages of the synchronous approach are:

a) Minimal overhead, since scheduling is pre-determined;

b) The scheduler is simpler, being essentially table driven;

c) The fixed schedule of task execution eliminates problems associated with code and data sharing, and does not require re-entrant code;

d) The load may be evenly distributed over the available time;

e) The deterministic behavior makes system verification easier.

The difficulties associated with it are:

a) It is difficult to structure programs so that they may be time-sliced;

b) Each time slice must be sufficient to accomodate the worst case, so on the average will be under-utilized;

c) It is difficult to accomodate response to random events such as crew inputs. Response to system failures is especially difficult, unless recovery from all classes of failures is pre-scheduled.

d) The structure is inflexible to change.

These disadvantages are all overcome by the demand scheduling approach, which, however, suffers from an increased degree of difficulty because of its greater complexity, and because it is more difficult to verify.

In a functional design of an executive for the Space Shuttle central computer, Intermetrics has proposed a combined synchronous and demand scheduled approach [6]. The repetitive, time-critical functions which can be implemented in short,

complete sections of code are executed by a synchronous "foreground" scheduler driven by timer interrupt, at 40 ms intervals. The majority of the remaining tasks are scheduled on demand as a "background" activity according to pre-assigned priority values. Communication between foreground and background is by an event mechanism, in essence similar to that described in section 2.3.2.

2.4.2 Deadlock Prevention

OS/360 has three resources to allocate to each job/step. These are core storage, data sets and peripheral devices. The allocation algorithm is summarized in Figure 3. Note that all data sets for the entire job are allocated at job initialization time and are bound for the duration of the job. In addition, all devices are allocated at step initialization time and are bound for the duration of the step. This approach may be costly since some of the resources allocated to a task may remain unused for long periods.

Alternatively, resources may be allocated dynamically, i.e., while the process is running. Unfortunately, now deadlock prevention becomes a more difficult problem. However, some practical solutions have been suggested [7], although a time overhead must be paid if they are implemented.

The suggested methods involve keeping track of the state of the system by means of state graphs or matrices. When a resource is requested by an executing process, the availability of the resource is checked. If it is presently unavailable, the algorithm must determine if it is safe to put the requesting task in the wait state. To determine this, it checks the state matrices of the system as they would be if the request were enqueued for the resource. When a safe condition results, the request is enqueued, and the task is placed in the wait state. On the other hand, if an unsafe condition results, the request must be denied and the task so notified. The task can then decide if it wishes to cease execution or it if can proceed without the resource. (Some subtle problems to be aware of, in implementing such an algorithm, have been overlooked by several authors and are discussed by Holt [8].)

While it is easy to see that dynamic allocation is most economical in the amount of time system resources are unavailable, some time overhead must be paid each time a process requests a resource. The OS must check the state matrices to determine if safe states will result. This process can be lengthy for a system with many resources and many ready tasks. One must remember here that the overhead is really that time

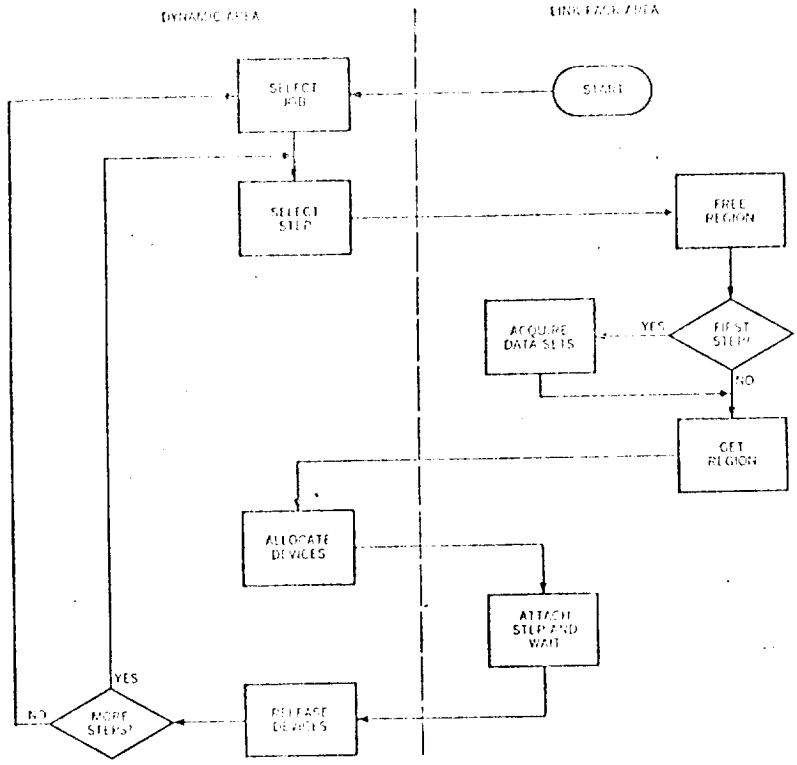


FIG. 3 : OS/360 Resource Allocation Algorithm

used for dynamic allocation over and above that which would otherwise be spent for allocation at job and step initialization times as described above.

Unfortunately, no analytic studies or simulations of these algorithms have been done to evaluate overhead costs. However, with careful thought given to the implementation of a dynamic algorithm, its overhead can be held to a minimum. In any case, the advantages of dynamic allocation would seem to overshadow any time overhead that results.

2.5 Memory Management

Management of the use of memory is potentially the most critical activity of an operating system. It is very dependent on:

a) the structure and characteristic behavior of the application software. If the work load is well known and dynamically predictable, especially with regard to its memory requirements, allocation of space can be predetermined, by pre-planned overlays for example.

b) The system architecture. If sufficient operating memory is provided to accomodate all programs at all times, dynamic allocation problems are eliminated. If, however, a virtual memory design is adopted for its potential simplification of programming and its cost effectivity, the operating system becomes intimately involved in creating and allocating memory space, and its detailed design is further affected by the technique adopted for addressing the virtual memory system.

c) Memory technology. The architecture of a virtual memory system and the functions of its operating system are significantly different for secondary storage with moving head disks, than for solid state block-oriented, random access devices such as the experimental magnetic bubble domain memory.

Although memory management can assume a critical role in determining operating system size and efficiency, its problems cannot be addressed in detail in the absence of a memory hierarchy definition. The following review of methods of operating memory utilization is presented to underscore some of the factors involved in providing increasing levels of operating memory utilization by multiplexing.

2.5.1 Operating Memory Multiplexing

The following examples describe practical applications of a number of techniques for increasing the utilization of operating memory.

2.5.1.1 Non-multiplexed Memory: In a non-multiplexed system the process of "assembly" of the program serves both to establish the mapping between names found in "subroutines" (which are simply separately maintained units of program code), and the mapping between names and physical locations in memory. At the conclusion of the assembly, the mapping information is completely distributed, and is saved and accessible only as a diagnostic aid, for the computer simulator, for example. Most flight control computers are of this design, usually because of their modest total memory requirements, typically 8K to 32K words.

2.5.1.2 Partitioned Memory: A simple form of memory multiplexing is used when the physical memory is large enough to support the requirements of more than a single program at a time. The OS/360 MFT and MVT systems implement fixed and variable partitions respectively. The normal objective of concurrently-loaded programs is to provide more efficient use of the processor by increasing the chances that some program can use the CPU when another is waiting for completion of I/O operations.

As in sequential execution, the mapping between names and locations is applied in all places at the time of loading, and the map is of no further use to the execution of the program.

To further increase processor efficiency, a high-speed secondary storage device may be used for "core-swapping". This involves writing the contents of a partition out to the device before its execution has been completed in order to make room to bring in some other program ready to run. Because the name-location mapping is not dynamically applied, the information must be returned to its original location when its execution is to be resumed.

2.5.1.3 Partitioned Memory with Relocation Registers: Under the above mechanization, the application of the name-location map takes place at one time, but over many spatial places. This has the advantage of getting the mapping finished; however, it has the disadvantage that the mapping is not readily reversed or modified. Several systems (e.g., PDP-10, Univac 1108) use an alternate scheme which re-applies the mapping each time. This

is achieved by providing one or more relocation registers, whose function is transparent to the software, which supply offset values to be combined with logical or virtual addresses generated during the program's execution. A disadvantage of this approach is that it requires additional hardware to perform the combining as part of instruction execution. However, it has the valuable characteristic that the mapping remains available for modification, so that program and data sections may be relocated in the operating memory and only the relocation values need to be changed in the process. Thus, storage in use can be compacted to collect available space into one contiguous piece when necessary to find room to load an additional program.

As in the partitioned memory scheme, "core-swapping" may be used for additional multiplexing. However, the use of the relocation registers makes it possible to return the information to any convenient location, rather than the precise place from which it was written.

2.5.1.4 Paging: An alternate to the use of relocation registers is to divide the program and data space, linearly arranged, into a series of "pages" of a fixed size, ordinarily a power of 2 (e.g., XDS Sigma 7, CDC 3800). In address formation, a group of bits from the logical address is used to select a page-location word from an array called a page-table; this word contains the memory-address of the page if it is currently there. Otherwise, an indication of the absence of the page is provided, along with the secondary-storage location at which the page may be found. The physical storage space is thus divided into fixed-size page frames, and the mapping between names and physical location is dynamically applied. A strong advantage of this approach is that logically contiguous space need not be physically contiguous, nor need it even all be present. The relaxation of the pages for occupancy of storage space by implementing some measurement of page reference behavior (with hardware help). Pages appearing to be less needed may be overlaid with more lively ones.

Because all page frames are the same size, space management is simple, and requires only modest overhead at execution time. On the other hand, the page boundaries fall at arbitrary locations in code or data, rather than at logical divisions. The average usefulness of words in a page is therefore reduced, since a logical entity may occupy only a small part of a page, or cross a page boundary.

2.5.1.5 Segmented Addressing: The simplest segmentation on a logical (if not operational) basis is the scheme used in the Burroughs B6700 and its predecessors. Each program block is compiled into a virtual address space of its own, called a segment; locations may then be accessed by specifying a segment number and an offset from the beginning of the segment. In execution, the name-location mapping is applied dynamically. Each segment has a segment descriptor which contains the physical location of the beginning of the segment. However, this descriptor can also contain an indication that the segment is not in storage at the moment; in this case, the address in the descriptor is the secondary storage location at which the segment may be found.

An advantage of this type of segmentation is the direct relationship between the segment size and the logical unit of program or data it contains. This characteristic increases the average usefulness of words transferred in a segment load.

A disadvantage of this scheme is that segments are small, segment descriptors are therefore numerous, and must consequently be located in operating memory rather than high-speed processor registers. The access to these necessarily slows down the address formation process; consequently some scheme of buffering in a small set of fast registers is usually utilized to shorten the access delay (see section 2.2.3). A second disadvantage is that storage allocation occurs in variable sized units and is therefore more complex and consumes more processor time than for fixed-sized pages.

2.5.1.6 Segmentation Plus Paging: This method of addressing and multiplexing was developed by the Multics group at MIT Project MAC. It is implemented most ambitiously on the GE (Honeywell) 645 designed for Multics, and also on the IBM 360/67. In Multics, segments tend to be large, and each is divided into fixed-size pages. Even page tables are paged, since they otherwise would occupy too much operating memory. Paging is the mechanism which accomplished multiplexing; segmentation is utilized for other purposes which are not relevant to this report. However, it should be mentioned that segmentation is implemented in such a way that when two independent processes refer to the same segment, both processes utilize the same page table. Sharing is thereby implemented in a general and powerful way.

The Intermetrics multiprocessor design [4] featured a segmented virtual memory system based in principle on the Burroughs designs. The policies for space allocation, segment placement, and replacement were, however, novel implementations of the operating system. The overall objective of the design

was to reduce the usual overhead consumed by the memory management function, by hardware assistance of address translation with associative memories local to the processors, and by specially tailored OS routines to handle segment I/O.

A more detailed examination of the characteristic differences between paging and segmentation, and the factors influencing virtual memory design is presented in Chapter 4.

2.6 Implementational Aspects

This review, far from complete, of multiprocessor operating system design problems closes with some comments about the implementational aspects. The major objectives of anyone embarking on the design of an operating system should be:

a) That the completed system work very closely to the way it was intended;

b) That it not take forever to finish;

c) That the resulting design be non-subtle, that it may be easily understood, maintained, and if necessary modified, other than by its creators.

2.6.1 System Specification

A big step towards accomplishing the first objective is to establish clearly in the beginning what the operating system is expected to do, and how. A considerable fraction of the total programming effort should be devoted to identifying the functional requirements, and then thinking out an overall, coherent design that not only satisfies them, but possesses enough flexibility to accomodate later modification and addition. The end item is a detailed design specification which deals with the structure to be implemented and its operating characteristics, and includes a description of how the completed system is to be verified.

2.6.2 Structure

The second and third objectives are largely a matter of the way in which the software of the operating system is structured, and the techniques used to implement that structure.

Comprehensive operating systems have acquired a bad reputation for complexity, cost and ultimate unreliability,

largely perhaps as the result of the widespread usage of the IBM 360 series of computers. OS/360 was very ambitiously conceived at a time when rigorous techniques of software construction (and the penalities of ignoring them!) were not as well researched and understood as today. Problems with the use of OS/360, and other designs, have prompted much study into the theory and practice of operating systems to be undertaken, especially during the last five years or so. A gathering body of knowledge on techniques of design and operation has become available (see, for example [9]).

Dijkstra has pioneered the disciplined approach to operating system design [10]. He organized the functions of a multiprogrammed operating system into a number of sequential processes. These processes were then hierarchically arranged to form several independent levels of increasing abstraction of machine operation. For example, the lowest hierarchical level was that of the real machine itself. At the next to lowest level were procedures for allocating processors to processes and fielding interrupts from the real time clock. The level above that managed the operation of the virtual memory, without concern for processor availability. The next level fielded the inputs from the operator keyboards, and so on. The application programs formed the highest level. A programmer was thus able to view the combination of hardware and software as a "virtual machine", representing an abstraction of the real machine. Needless to say, the whole concept precluded the use of machine language coding by any application programmer, since this would have cut straight through the screening levels of "virtual machines". Each level of the system possessed a large degree of independence of the other levels, and could be separately conceived, implemented and tested.

Other operating system designs with different operational requirements and system configurations would probably depart from the functional separations made by Dijkstra, but the basic philosophy may be adhered to.

2.6.3 Systems Programming Language

Just as a problem oriented higher level language assists in the structuring and implementation of applications software, the use of a language suited to the definition of OS functions has gained much support from operating system implementers. The advantages can be viewed from both a managerial and a technical aspect. The managerial benefits of HOL usage are too well established to be repreated here. Various authors have defined the features that would make a systems programming language easy and efficient to use [11]. Almost all agree that

the language should possess a block structure and enforce name scope rules. It should contain control features such as procedures and functions, the statements IF THEN ELSE, DO FOR, and DO CASE. Some language designs restrict data types to those generally agreed to be useful to systems programming, namely bit, character, pointer and various forms of arrays. Others, following the example of PASCAL [12] contain more powerful and flexible data structures, which allow the systems programmer freedom to adapt the language to his specific problem. The ability to address specific machine features is necessary, although the major portion of any operating system can be machine-independent. The need to generate efficient code is clear, if only to overcome the reluctance of non-believing systems programmers to code in a higher level language! Almost all advocates insist on the absolute necessity of readability in the language, and the provision of comprehensive diagnostics by the compiler. From these characteristics, it is evident that systems and application programming languages have quite similar objectives, and differ mainly in the natural incompatibility of the data types recognized. Several attempts have been made, therefore, to adapt existing HOLs for system programming, as the following examples illustrate.

A subset of PL/I was chosen to code the operating system for the comprehensive Multics system at MIT [13], which is based on Honeywell 6000 computers. The Burroughs Corporation has developed several versions of ALGOL 60 with differing degrees of machine dependence [14] for different B6700 systems programming applciations, as a consequence of their long standing use of ESPOL in the B5500. There is Extended ALGOL for the bulk of systems programming, including the Extended ALGOL compiler itself; Data Communication ALGOL, which allows the control software for communications interfaces to be conveniently programmed; and ESPOL, the original systems language, which enables many of the B6700 features such as stacks, registers, memory, the multiplexors, peripheral devices, etc., to be addressed directly.

Several languages have been developed to handle systems programming for specific machine architectures. The University of Toronto is developing SUE for system programming on the IBM 360 [15]. An extensible language LSD is being designed for systems development on the IBM 360 at Brown University [16], although it is not yet operational. PL 360 [17] a language designed by Wirth at Stanford University for the IBM 360, has features that make it attractive to systems programming. Carnegie-Mellon has developed and used BLISS for its DEC PDP-10 [18].

It is strongly recommended that the operating system for the SUMC is designed and written in one of these systems programming languages, or at least in some tailored subset.

Most of the compilers have been written in the language itself, which lessens the difficulty of transferring the compiler from its original host machine to another.

References for Chapter 2

1. Fulmer, L.C., "A Modular Plated-Wire Associate Processor", GER-14727 Goodyear Aerospace Corp., Akron, Ohio, March 1970.

2. Conti, C.J., "Concepts for Buffer Storage", Computer Group News, March 1969.

3. Entner, R.S., "The Advanced Avionic Digital Computer System", Computer Design, September 1970.

4. Intermetrics, Inc., "Engineering Study for the Functional Design of a Multiprocessor System", Final Report, NASA/Intermetrics Contract NAS9-11745, September 1972.

5. Intermetrics, Inc., "The Programming Language HAL - A Specification", NASA/Intermetrics Contract NAS9-10542, June 1971.

6. Intermetrics, Inc., "Advanced Software Techniques for Data Management Systems: Vol. II", Final Report, NASA/Intermetrics Contract NAS9-11778, February 1972.

7. Coffman, E., et. al., "System Deadlocks", ACM Computing Surveys, Vol. 3, No. 2, June 1971.

8. Holt, R., "Prevention of System Deadlocks", Comm. ACM, January 1971.

9. ACM/SIGOPS, "Operating Systems Review", Proc. 3rd Symposium on Operating System Principles, Stanford University, Palo Alto, California, October 1971.

10. Dijkstra, E.W., "The Structure of 'THE' Multiprogramming System", Comm, ACM, Vol. 2, No. 5, May 1968.

11. ACM/SIGPLAN, Proceedings of Symposium on Languages for Systems Implementation, Purdue Univ., Lafayette Ind., October 1971.

12. Wirth, N., "The Programming Language PASCAL", Acta Informatica 1, 35-63, (1971) by Springer Verlag, 1971.

13. Corbato, F.J., "PL/1 As A Tool for System Programming", Datamation, Vol. 15, No. 5, May 1969.

14. Lyle, D.M., "A Hierarchy of High Order Languages for Systems Programming", Proc. Symposium on Languages for Systems Implementation, October 1971.

15. Clark, B.L., et. al., "System Language for Project SUE", Proceedings of Symposium on Languages for Systems Implementation, October 1971.

16. Bergeron, R.D., et. al., "Language for System Development", Proc. Symp. on Languages for Systems Implementation, October 1971.

17. Wirth, N., "PL360, A Programming Language for the 360 Computers", Journal ACM, Vol. 15, No. 1, January 1968.

18. Wulf, W.A., et. al., "Bliss Reference Manual", Carnegie-Mellon University, Pittsburgh, Pennsylvania, January 1970.

Chapter 3

INTERRUPT STRUCTURE

This chapter will discuss various aspects of the interrupt structure when applied to a multiprocessor. The first section will present a list of assumptions upon which the following sections are based. The second section presents a brief categorization of interrupts expected within the environment of the space station multiprocessor. The third section discusses various problems that are encountered when attempting to develop an interrupt structure for the multiprocessor.

3.1 Assumptions

a) The basic assumption is that the concept of interrupts is indeed required. It is possible to conceive of computer systems that are well specified, in which all equipments are synchronized and serviced in a predetermined cyclic fashion. However, the system contemplated for the space station is not well specified. It will have to respond to conditions not anticipated in the program flow. Therefore, the need for interrupts is postulated.

b) A true multiprocessor is assumed. This includes a "floating executive" and a configuration with three or more processing units. With a floating executive, any process can be executed on any processor. There are no functions dedicated to any processor. This excepts the I/OC, which does serve a specialized function. Three or more processors are assumed so that the generalized solution to multiprocessor interrupt handling can be addressed.

3.2 Interrupt Categorization

An interrupt can be defined as any condition which causes an involuntary interruption in the sequence of execution of a process. The interrupt is not explicitly anticipated in a program's code. It can be considered to be an involuntary procedure call to the interrupt servicing routines, with an ultimate return link to the original process.

Interrupts may be categorized into three distinct classes:

3.2.1 Process Oriented

Process oriented interrupts are those associated with the process in execution. There are a number of distinct types. Arithmetic and control traps are caused whenever an unacceptable condition presents further execution. An interrupt from a "watchdog timer" indicates that a process has been running for an excessive time.

The above two process-oriented class of interrupts are synchronous with the process and occur while the process is running. There exists a class of process-oriented interrupts which can occur when a process is in a waiting state. These interrupts, sometimes called software interrupts, result from HOL statements of the following form, as discussed in section 2.3.2:

ON (event) <code block>

This statement establishes a linkage which causes the specified <code block> to be executed when the specified (event) is signalled. If the process is running when the (event) is signalled, then it is interrupted to execute the <code block>. If the process is not running when the (event) is signalled, then as soon as the process which issued the ON statement enters the running state it will be interrupted to execute the <code block>.

3.2.2 System Oriented

This category of interrupt does not have any particular affinity for the currently running process. Conditions such as I/O Complete, I/O Error, and Absent Segment Trap fall into this category. Both I/O Management and Memory Management are executive functions.

Many failures or error conditions, such as power failure, can be considered system oriented.

3.2.3 Processor Oriented

Even with a floating executive and no dedicated functions to particular processors, it does become necessary to direct an interrupt to a specific processor, independent of the process being executed. For example, in response to an error signal one processor might direct another processor to terminate or restart. The entire area of system initialization

and reconfiguration requires direct communication with specific processors. The processor-directed interrupt is a convenient mechanism for meeting this requirement.

3.3 Multiprocessor Interrupt Problem Areas

A number of problems involved in the servicing of interrupts exist. Some are aggravated in a multiprocessor environment and some are unique to the multiprocessor environment. Four major areas are discussed below.

3.3.1 Which Processor to Interrupt?

In a multiprocessor system, a question arises as to which processor to select to handle a given interrupt. For process oriented interrupts which occur while the process is running, the decision is trivial. The interrupt should be steered to the related processor. Similarly, so should processor related interrupts.

The remainder of the interrupts are system-oriented or non-running-process-oriented, and have no affinity for any particular processor.

A number of options are possible in assigning a processor to service the interrupt:

a) An arbitrary processor may be interrupted based upon some random selection algorithm. The interrupted processor may then execute a software routine which determines whether the interrupt condition is of higher priority than the process which was interrupted. If it is not, then the interrupted process will be scheduled like any other process according to its priority.

b) All the processors may be interrupted. The interrupt service routine can be made a "critical section" of code which can only be executed by one processor at a time. The first processor to access this code services the interrupt. The other processors revert to their original processes.

c) A sequential selection employing a "round robin" style algorithm may be used. In this way, the interrupts are loaded equally upon all processors. This option of course does not consider the process which is running on the processor at the instant of interruption.

d) An assigned processor might service all interrupts, or specific interrupt conditions can be preassigned to specific processors.

e) The processor executing the lowest priority process will be selected to service the interrupt. If the interrupt priority is lower than any running process, then the required interrupt response will not be executed until a process swap results in a lower priority process.

The approach recommended in this Report is to provide a combination of c) and d) by placing within the I/O control an element of hardware which automatically determines the most interruptable processor (based upon the priority of the process running), and receives and distributes all potential interrupts.

Running-process-oriented interrupts can by-pass the interrupt logic within the I/OC since the processor to be selected is known a-priori.

3.3.2 Response Time

There is a small class of interrupts which require almost immediate response. These are system oriented and deal with equipment failures or other emergency situations. One example is a "power failure" interrupt. This must be responded to within microseconds in order to move any volatile registers into permanent storage and then systematically to shut down the system.

The class of conditions associated with arithmetic and control traps does not require instantaneous response but the running process can not continue until after the trap is serviced. Any trap condition falls into this category, even system oriented traps such as the Absent Segment Trap.

Quite often specifications are generated and systems built which require I/O Complete interrupts to be generated within micro seconds of an I/O completion. From a performance point of view, most I/O interrupts can possess a response time of the order of milliseconds. For example, if M3 requires an average of 10 milliseconds for each access, it is clearly unnecessary for its completion to be signalled within microseconds.

3.3.3 Innovations

A number of innovations may be suggested in the I/O interrupt area. These suggestions exploit the space station type of I/O, namely mass storage M3, and a data bus.

a) "Quiet" I/O

When the multiprocessor system workload is heavy, the frequency of Absent Segment Traps can be expected to be relatively high. Conventional processing of an Absent Segment Trap requires entry to an interrupt handler, initiation of an M3 operation, and placement of the process into the wait state. Upon completion of the M3 operation, an I/O Completion interrupt is signalled. The handler for this interrupt is then entered, the process waiting for the segment is readied, another I/O operation to M3 is initiated if one is queued, and the processor allocation routine is called to see if it is appropriate to assign a processor to the newly readied process.

An alternate implementation is suggested to avoid, at least in most cases, the necessity for entering the I/O Completion interrupt handler when the segment transfer is concluded. This is achieved by providing a capability in the I/O controller which causes it to make a choice of whether or not to signal I/O completion. Thus a dynamic decision is made as to whether the interrupt should be suppressed or signalled, depending upon the existence of a queue of operations waiting for the device. If the interrupt is suppressed, the condition is made known to the system by the setting of a bit field in a location accessible to the absent segment trap handler. After initiating the M3 operation to make an absent segment present, this handler checks the completion-states of M3 segment transfers previously issued. The processes whose segments are found to have completed their transfers are readied; thus the utilization of the I/O Completion interrupt handler is avoided. This diminishes the overhead for absent segment handling,.especially under heavy load, when computational overhead is most detrimental to the system throughput.

b) Data Bus Control

If a command response data bus, with a minor cycle of 20 milliseconds, is employed then it is clearly unnecessary to interrupt the system after each peripheral device is accessed. In principle, the synchronous nature of the data bus does not require interrupts for normal processing. However, one may consider the need for interrupts due to infrequent events:

1) An interrupt might be generated by the Data Bus Control Unit if certain types of failures are detected.

2) For equipments which are interrogated at a very low frequency or even randomly an interrupt might be considered at the end of the request.

Both of these suggestions impose little if any load on the system due to their very low frequency of operation. A checkout problem may, however, arise in trying to verify successful operation for infrequent interrupts at any point of execution in a program.

3.3.4 The Interrupt Sequence

When an interrupt is signalled to a processor, the details of its local environment, the processor's status, must be saved so an eventual return is possible. In a stack oriented machine an interrupt response can be executed parasitically on top of the process' stack, with entrance and return functions performed automatically.

Since procedures may be nested to multiple depth, so can interrupts. The only limit is the number of display registers. provided to mark the beginning of each lexical level in the stack.

3.3.5 Interrupt Functional Response

System or processor-oriented interrupts possess a static (pre-determined) response. Once the response is established it is not changed. However, for process-oriented interrupts (traps) one may conceive of situations where each process may desire a different response to particular interrupts. For example, one process might want to respond to a square root of a negative number trap by substituting a zero for the answer. Another process might deal with complex numbers and cause a re-entrance into the square root instruction with a change of sign of the argument.

For all trap conditions the system must provide a default option. It is suggested that a process be allowed to override this system option by providing its own response to particular traps. Any process at any lexical level should be allowed to specify, if necessary, its own response to process-oriented traps.

Chapter 4

MEMORY HIERARCHY

Memory is possibly the most difficult of any computer element to specify, implement and use. It is in this area that technological limits and cost factors are first encountered when considering the design of an advanced high performance computer system. The inability of a single, currently known, memory technology to meet the conflicting requirements of high access speed and high storage capacity has led to the hierarchical concept of levels of memory.

4.1 Basic Hierarchy Description

Within the multiprocessor structure, one finds a number of levels of memory used for varying purposes.

4.1.1 M0 - Micro Level Control Memory

From one point of view, micro memory is only a particular implementation of a control unit and therefore should not be considered part of the memory hierarchy. Alternatively, another point of view suggests that micro memory should be used for execution of the frequently used operating system primitives and subroutines. It is from this secondary point of view that M0 is considered an element of the memory hierarchy.

4.1.2 M1 - Local Memory

M1 storage is dedicated to the processing unit. Its function can range from a register set, as is found in the SUMC, to a complete cache memory as used in the IBM 360/85. The major function of M1 is to increase the performance of the system. Its speed is in the 100 nanosecond access time range and its size can range from 16 words (for register storage) to 4K words (for a cache implementation).

4.1.3 M2 - Operating Memory

In a multiprocessor environment, M2 is that part of memory which is shared by the processing units and I/O controllers.

M2 must of necessity consist of a number of separate memory modules so that simultaneous access of different modules may be made by the processing units and IO/C. M2 cycle time is in the 1 microsecond range and its size is of the order of 100K words.

4.1.4 M3 - Mass Memory

M3, historically a drum or disk, provides the function of augmenting the M2 storage. It is used to hold all the programs and data segments not currently being used in the processing function. M3 is used to implement the concept of a larger M2 virtual memory. It is characterized by an access time in the millisecond range and a storage size consisting of millions of words.

4.1.5 M4 - Archival Storage

Archival storage (possibly implemented with a magnetic tape unit) is included for completeness. It is used as the repository of files and other information which does not undergo rapid change or frequent use. Conventionally, M4 is considered to be an I/O device and is controlled accordingly.

The remainder of this chapter will concentrate on the relationships between the major elements of the memory hierarchy which contribute to system performance, namely M1, M2, and M3.

4.2 Local Storage

4.2.1 The Problem - Memory Contention vs. Performance

One of the major reasons for using a multiprocessor is to increase the overall performance or work delivered by the system. If the extra performance were not required a uniprocessor would be employed. Ideally, a system with R processing units should produce R times the work of a single processor system. One factor which tends to reduce the overall performance of the multiprocessor is M2 memory contention. The effect is to reduce the M2 cycle time (t_2) by yielding an effectively slower cycle time (t_{2eff}).

One way to reduce memory contention is to provide a limited amount of dedicated memory local to each processor (M1). If M1 possesses a cycle time (t_1) which is substantially faster than t_2 then a performance increase can be obtained.

4.2.1.1 Performance Model: Postulate the multiprocessor model shown in Figure 4.1, and make the following definitions and assumptions:

a) n_1 = number of M1 cycles per unit time for a single processor

b) t_1 = M1 cycle time

c) n_2 = number of M2 cycles per unit time executed by a single processor

d) t_2 = M2 cycle time

e) t_{2eff} = effective reduced M2 cycle time due to memory contention

f) W = work per unit time from a single processing unit. This is defined as proportional to the total number of M1 and M2 cycles per unit time. Usually processor work is defined in terms of the number of instructions per second. For a conventional 360 type architecture an instruction usually corresponds to two M2 cycles. In a sense the internal processor cycles should also be considered useful work. Indexing which does not require an M2 access, because it might use an internal register is a very useful function. If a multiprocessor makes very large use of its internal M1 storage these cycles are just as important as M2 cycles in estimating overall work.

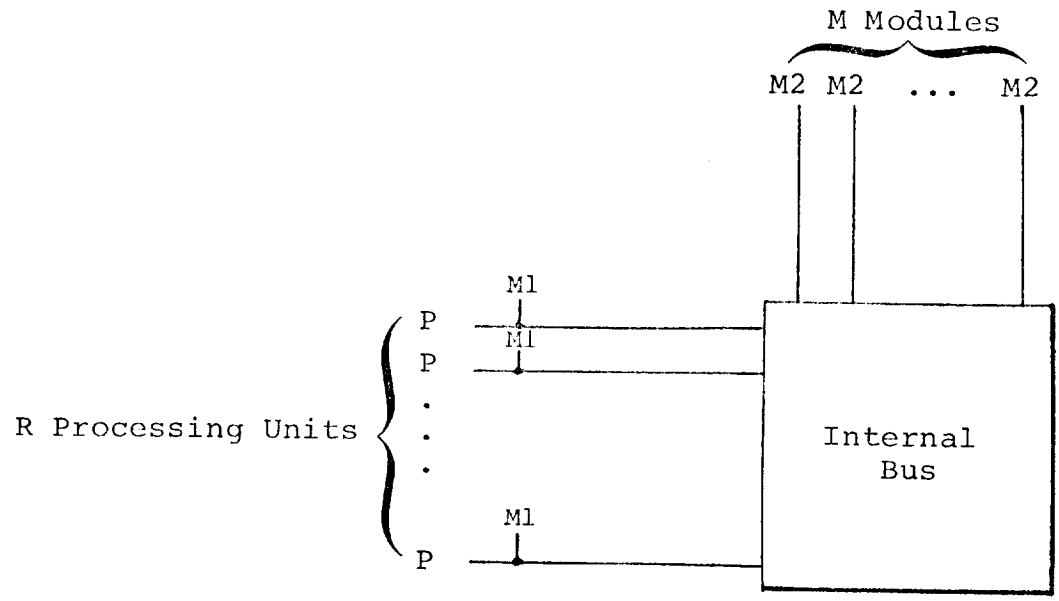
$$W = n_1 + n_2$$

g) R = number of processing units

h) M = number of independent M2 modules

i) h = fraction of all memory requests that use M1 (the hit ratio). This is for a single processing unit.

$$h = \frac{n_1}{n_1 + n_2}$$



Notes:

1) A processing unit contains a P-M1 combination

2) The internal bus allows all the R processing units to communicate with all M memory modules

3) There is no internal bus contention

Figure 4.1: Multiprocessor Model

j) It is assumed that a processing unit is always making an M1 or an M2 reference and that these references are mutually exclusive, that is they cannot occur simultaneously. Let $n_1 t_1 + n_2(t_{2eff}) = 1$ unit of time. From the above definitions it follows that

$$W = \frac{1}{t_{2eff}} \left[\frac{r}{h + (1 - h)\, r} \right]$$

where $$r = \frac{t_{2eff}}{t_1}$$

The term in brackets can be considered to be an enhancement factor by which performance is increased. Figure 4.2 plots this factor as a function of h.

We see from this simplified model that the introduction of M1 with a reasonably high hit ratio can potentially increase the performance of a processing unit, especially if the t_2/t_1 ratio is high. Many overhead factors, involved in the utilization and control of M1 will tend to lessen the improvement.

The effect of memory contention upon t_{2eff} will now be calculated. Assume that requests to M2 are independent and randomly distributed across the address space. In reality this assumption can be seriously questioned since program and data both possess locality. That is, there is a strong correlation between successive M2 access events. This is extremely difficult to measure since the programming load is not known. For lack of a better model, the random distribution is assumed.

A processor will request access to M2 with a probability $A = n_2(t_{2eff})/n_1(t_1) + n_2(t_{2eff})$. It can be shown that

$$A = \frac{r(1 - h)}{r(1 - h) + h}$$

The probability of accessing any particular M2 modules is therefore A/M. Given that a processor is requesting access to a particular M2 module, the probability that none of the other R - 1 processors are requesting access to that module is:

$$k = \frac{r}{h + (1-h)\ r}$$

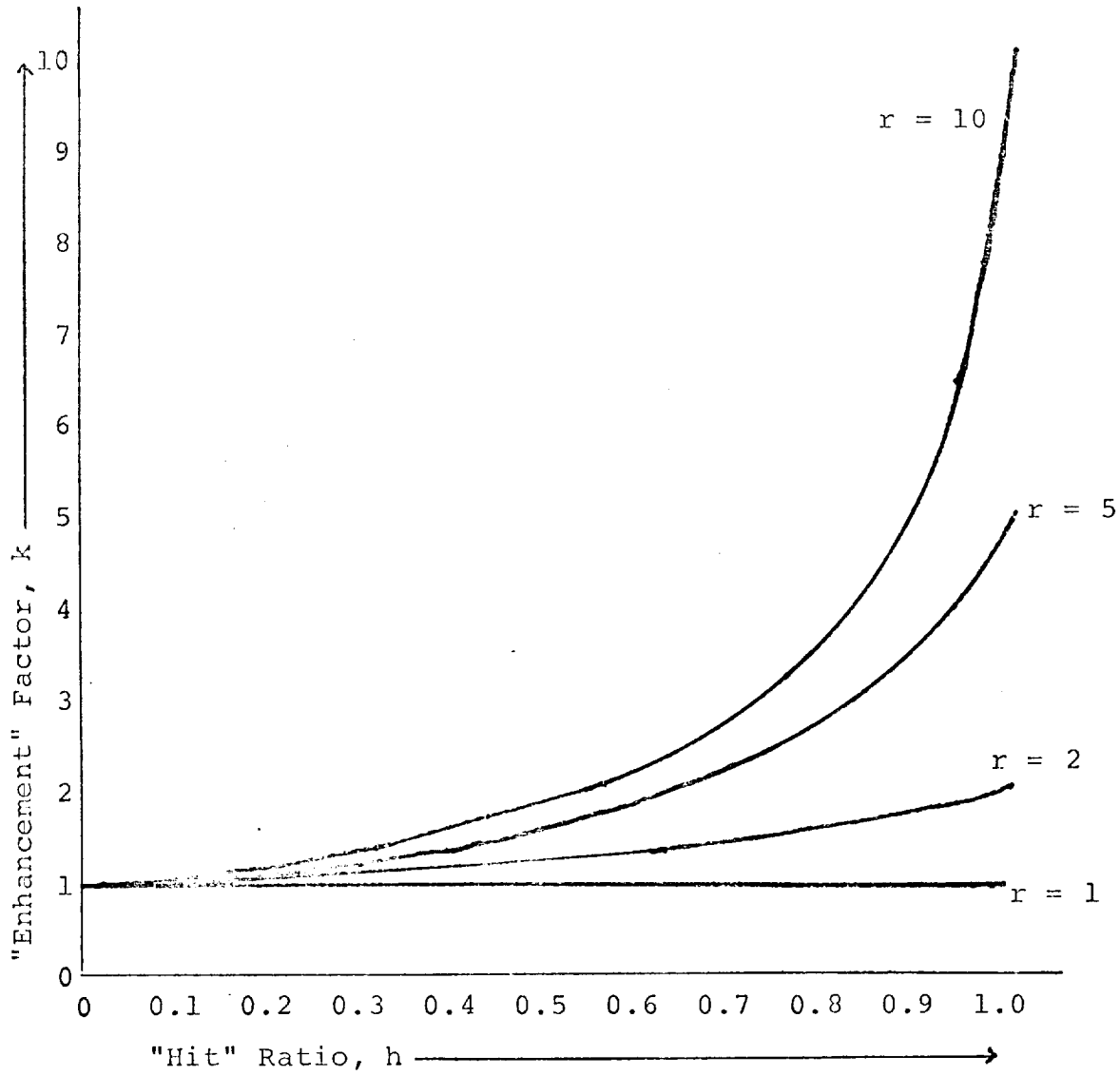


Figure 4.2: "Enhancement" Factor k Versus "Hit" Ratio h

$$P(0) = (1 - A/M)^{R-1}$$

The probability that 1 out of R-1 other processors is requesting access to the particular M2 module is:

$$P(1) = \binom{R-1}{i}^{*} (1 - A/M)^{R-2} (A/M)^{1}$$

In general, the probability that i processors out of the R-1 other processors desire access to the same module is:

$$P(i) = \binom{R-1}{i}^{*} (1 - A/M)^{R-1-i} (A/M)^{i}$$

If there is no contention, the M2 access time is t_2. If one other processor is requesting, the access time could reach $2(t_2)$. In general, with i other processors the access time could reach $(1 + i)t_2$.

The effective access time averaged over all contention possibilities is therefore

$$t_{2eff} = \sum_{i=0}^{R-1} (1 + i)(t_2)P(i) = t_2 \sum_{i=0}^{R-1} P(i) + t_2 \sum_{i=0}^{R-1} iP(i)$$

Since Pi is a binomial distribution [1]

$$\sum_{i=0}^{R-1} P(i) = 1$$

and

$$\sum_{i=0}^{R-1} iP(i) = (R - 1)(A/M)$$

* $\binom{R-1}{i} = \frac{(R-1)!}{i!(R-1-i)!}$

therefore

$$t2eff = t_2 \left[1 + \frac{(R-1)A}{M}\right] = t_2 \left[1 + \frac{(R-1)\ r\ (1-h)}{M[r\ (1-h) + h]}\right]$$

Some insight may be gained by studying the overall total system work (W_T) where:

$$W_T = RW$$

$$W_T = t_2 \left[\frac{Rr}{[r\ (1-h) + h] + \frac{(R-1)}{M}\ r\ (1-h)}\right]$$

The following figures (Figure 4.3 and Figure 4.4) depict W_T for h = 0, h = .5. The following two facts should be observed:

a) System performance is increased as more M2 modules are added.

b) Local storage can significantly increase performance.

No Ml
h = 0
t_2 = 1 μsec

$$W_T = \frac{1}{t_2}\left[\frac{MR}{M + R - 1}\right]$$

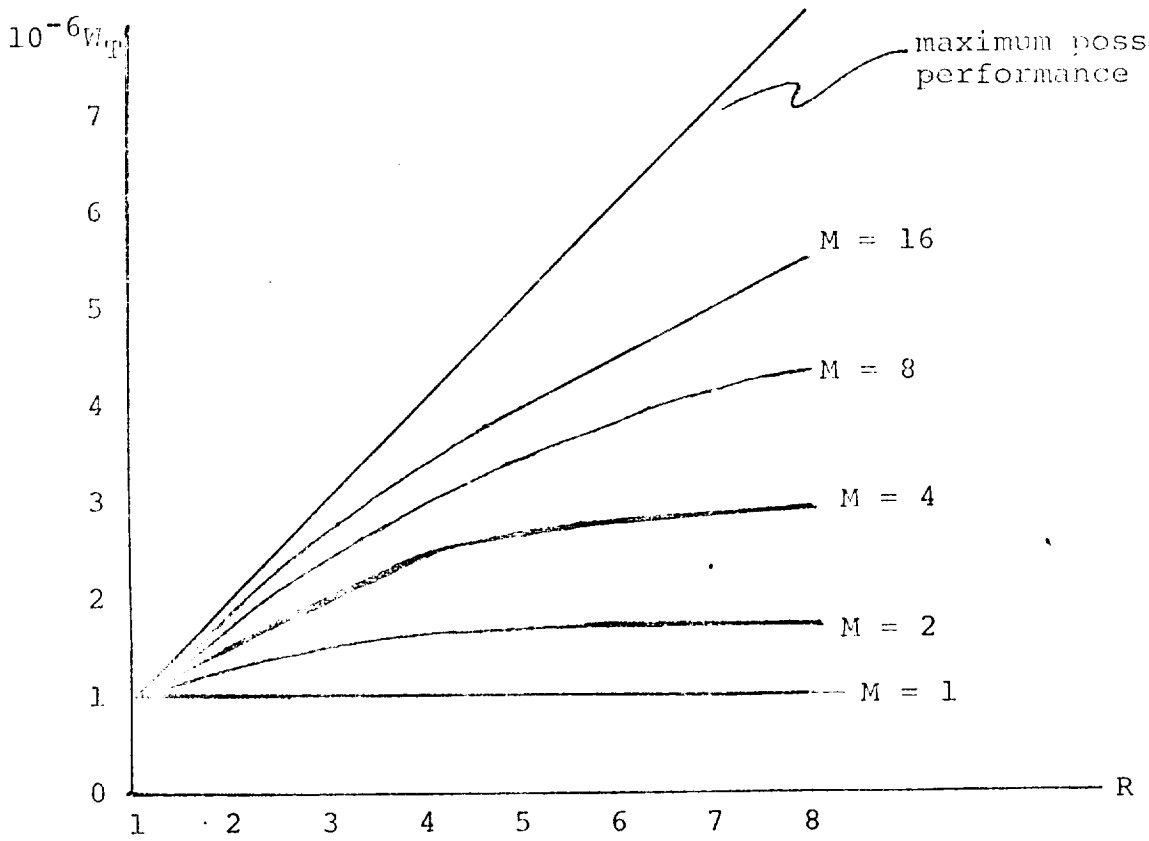


Figure 4.3: W_T Versus R

h = .5
r = 10
t_2 = 1 μs

$$W_T = \frac{1.82R}{1 + \frac{.98\ (R - 1)}{M}} \frac{1}{t_2}$$

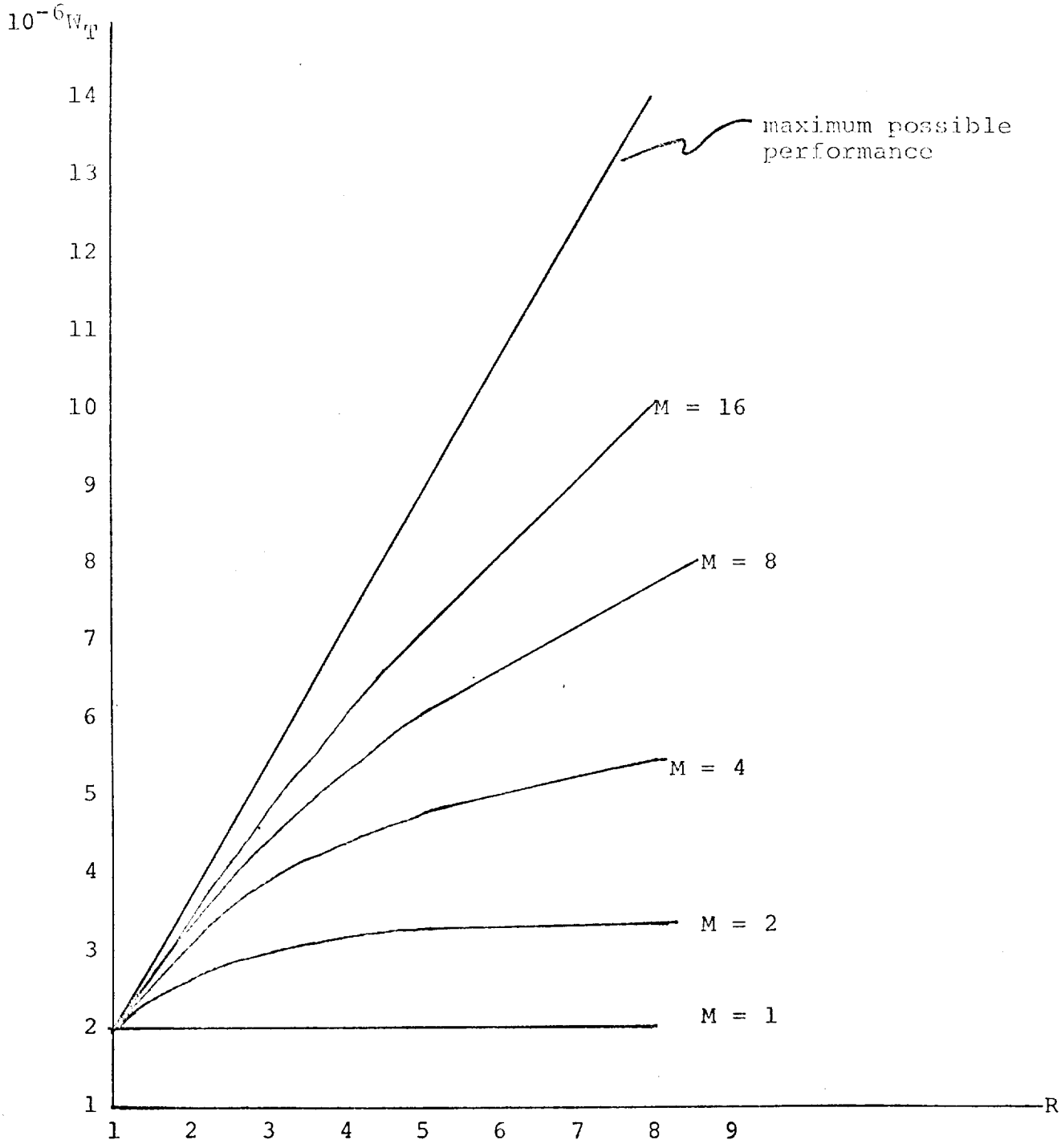


Figure 4.4: W_T Versus R

4.2.2 Two Approaches to an Implementation

A major design question naturally arises. How does one use local storage to obtain a hit ratio of .5 or .9 or more? The answer is complex and involves studying the nature of program execution in relationship to the instruction set. Two approaches will be mentioned.

4.2.2.1 The Cache Concept: As CPU speeds have increased with advances in technology, computers have been able to handle larger and more complex processing tasks, and the demand for operating memory capacity has increased. Since capacity and speed are conflicting factors in memory design, an hierarchical memory organization was proposed many years ago [2] to enable these two desirable qualities to be independently developed. Advances in semi-conductor technology have only recently made this concept feasible.

A backing store, M2, which de-emphasizes speed to achieve an adequate capacity, interfaces to a buffer store or cache, M1, whose primary design objective is speed.

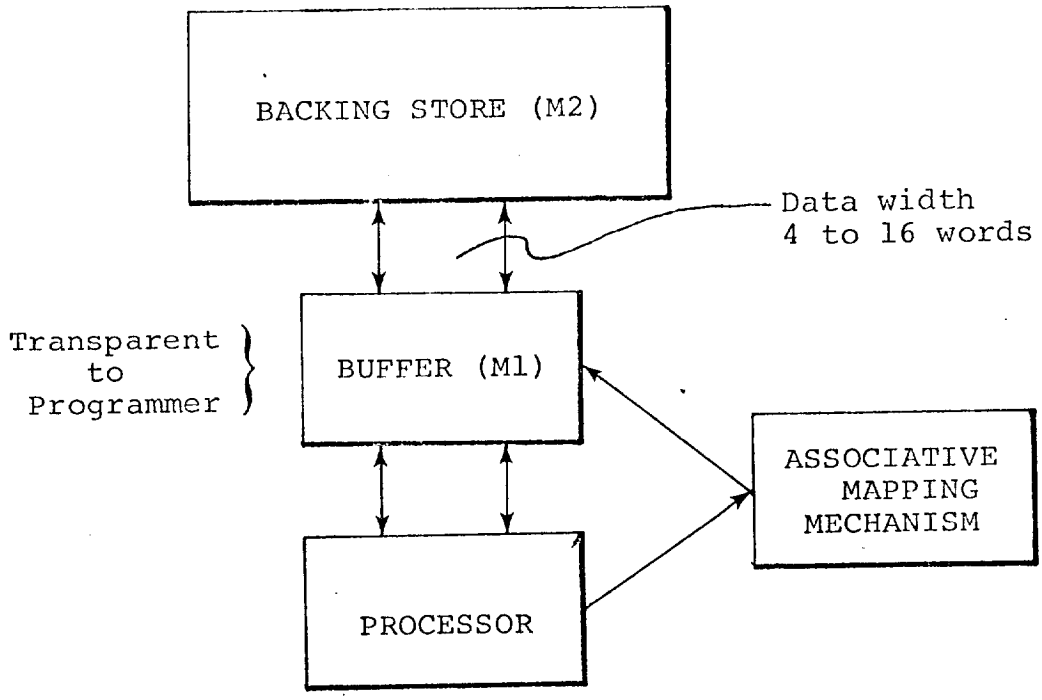


Figure 4.5: Buffer Store Organization

The concept depends for its success on the notion of locality. Locality is an experimentally observed fact of program behavior by which references tend to occur within a region of the program's address space, and this region migrates relatively slowly. Locality is a natural outcome of the way people think and write programs: concentrating on one task at a time, using loops, using sequential control, etc. [3]. The degree of locality is influenced by programming style, data organization, strategy of algorithm, and the programming language. Locality gives rise to the notion of the program working set, which is the minimal set of blocks that a program requires to have in the cache in order to run efficiently. If less than the working set is in M1, the probability of occurrence of a reference to a missing block, m, increases. This situation is most likely to occur in a multiprogrammed environment when the number of programs n exceeds the capacity of the cache to contain all their working sets, as illustrated below.

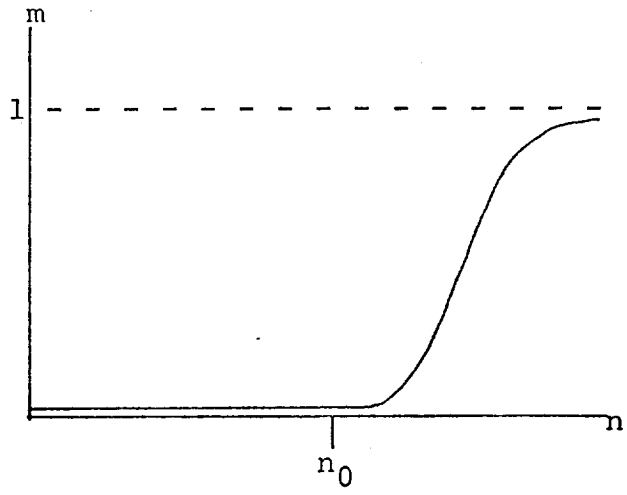


Figure 4.6: Probability of Missing Block Versus Number of Working Sets

It is an experimentally verified fact that a process favors references to a small set of its total address space, and that provided this set is contained within M1, the need to access program areas not in M1 arises relatively infrequently. When access to M2 becomes necessary, more information

than is immediately required is transferred to M1, in the expectation that references in the vicinity of the accessed word are likely. The relationship between the size of M1, the amount of information transferred, and the effect of different program addressing behaviors was studied by Gibson [3]. He concluded that an M1 capacity of 2K to 4K words and a transfer block size between 4 and 16 words provided best results. He also found that the dynamics of buffer operation were more sensitive to the addressing patterns of the various programs than to any other factor.

To maintain a given processor's speed, data transfer from M2 must occur at an adequate rate. The M2's slower access can be compensated for by increasing the transfer path width. This can be achieved by:

a) An M2 technology which yields a long physically stored word, e.g., the pseudo - 2 1/2D organized plated wire memory [4] which allows several hundred bits to be accessed at once.

b) Organizing M2 into a number of smaller modules and interleaving the addresses, so that contiguous addresses 1, 2, 3, are stored at corresponding locations in modules 1,2,3, rather than in conseuctive locations in any one module. This has been the approach employed by current designs such as the IBM 360/85, 91 and 195, which use core technology for M2.

The high speed of M1 is now generally realized by bipolar semiconductor techniques rather than thin-film. Buffer memories of up to 1/4 million bits with cycle times less than 200 ns have been built, although similar speeds at far lower power dissipations are being achieved by current plated wire designs [5].

The above discussion has been in terms of a processor "read" operation. Writing into the buffer presents an additional problem in that the contents of the buffer do not represent the primary source of the program being processed. A processor "write" must be reflected in an update of the primary source, which is stored in M2. This can be achieved in two ways:

a) Storing through: Every "write" request causes an immediate update of M2 as well as the cache.

b) Block update: Write requests are allowed to accumulate in M1. Whenever a block is to be replaced by the block replacement logic the modified block is written out to M2.

Which of the two techniques is chosen depends on program behavior: "writes" tend to cluster in time and in program space, so that for small blocks of 4 to 16 words a block update technique may result in lower average M1 to M2 traffic density and transfer delay.

There are a number of arguments which can be raised against the use of a cache in a multiprocessor system.

a) Cost. To be effective a 4K word cache of high speed (100 μs) monolithic memory must be employed in each processing unit.

b) To keep the cache filled with useful data a large bandwidth of data from M2 must exist (128-256 bits) per access. Many of the words accessed from M2 might not be used. This unnecessary M2 traffic tends to increase M2 contention and thus reduce performance.

c) In a multiprocessor system the use of a cache with COMPOOL data presents a problem in keeping copies, present in the various caches, updated. (See section 2.3.1)

d) IBM's successful use of the cache is based partially upon the inefficiency of the 360 instruction set. That is, quite often small program loops are inserted by a compiler to execute primitive functions which could have been basic instructions in other systems.

4.2.2.2 M1 in a Stack-oriented, Descriptor-based System: The problem faced in employing M1 is to use it for information which has a high probability of being accessed many times (a high hit ratio, h). Traditionally, base registers and index registers have been allocated to the local storage of a processor for reasons of speed and their high frequency of use. However, register management problems tend to increase overhead.

Intermetrics proposes to use M1 for specialized storage and to have the management of M1 an automatic hardware function.

In a stack-oriented machine it was realized that the top few entries of the stack provide the most referenced elements. For this reason the first 8 stack locations are made resident in M1. M1 stack overflow pushes the bottom of the M1 portion of the stack into M2.

The descriptor is the most referenced data type. For this reason the 32 most recently referenced descriptors are retained in M1. An associative mapping mechanism is employed for control of this descriptor cache.

The dynamic nature of the stack creates a situation where the starting location of each lexical level must be quickly accessed. For this reason a set of from 16-32 base registers is proposed. Each base register contains a pointer to the start of each lexical level and is automatically accessed when addressing within the stack is desired.

An instruction set which is organized around this machine tends to be more complex than a 360 type instruction set. For this reason more time is spent accessing M1 and executing micro code. This tends to make the duty cycle of the processor higher than a 360 type instruction set, which in turn tends to reduce memory contention.

The processor's duty cycle and the parameter h are directly related.

$$D = \text{duty cycle} = \frac{n_1(t_1)}{n_1(t_1) + n_2(t_{2eff}}$$

$$h = \frac{n_1}{n_1 + n_2}$$

$$D = \frac{h}{(1 - h)\ r + h} \quad \text{where } r = \frac{t_{2eff}}{t_1}$$

4.3 Operating Memory and Memory Management

The concept of a memory hierarchy, discussed in relation to M1 and M2, can be extended to the relationship between M2 and M3. For large file oriented systems archival storage, M4, is also considered.

4.3.1 Background

Since program and data can only enter the computation process via M2 one must control the flow of information across the hierarchy of memory. This control is the job of memory management.

Virtual memory is a technique for managing the utilization of memory in processing systems where program space

exceeds the actual operating memory space. The concept has evolved from the need to improve on early attempts to utilize limited amounts of memory by overlaying. This required the user to partition his program into pieces which fit into the available space, and then plan the sequence of execution of the pieces and control their reading into and out of operating memory. As program requirements grew larger than a few thousand words, this became a cumbersome task. To help the programmer, automatic overlaying (folding) techniques, by the operating system with compiler assistance, were developed. But eventually it became clear that a system should allow a distinction to be made between address space, a set of identifiers used by a program to reference information, and memory space, the set of physical operating memory locations [6].

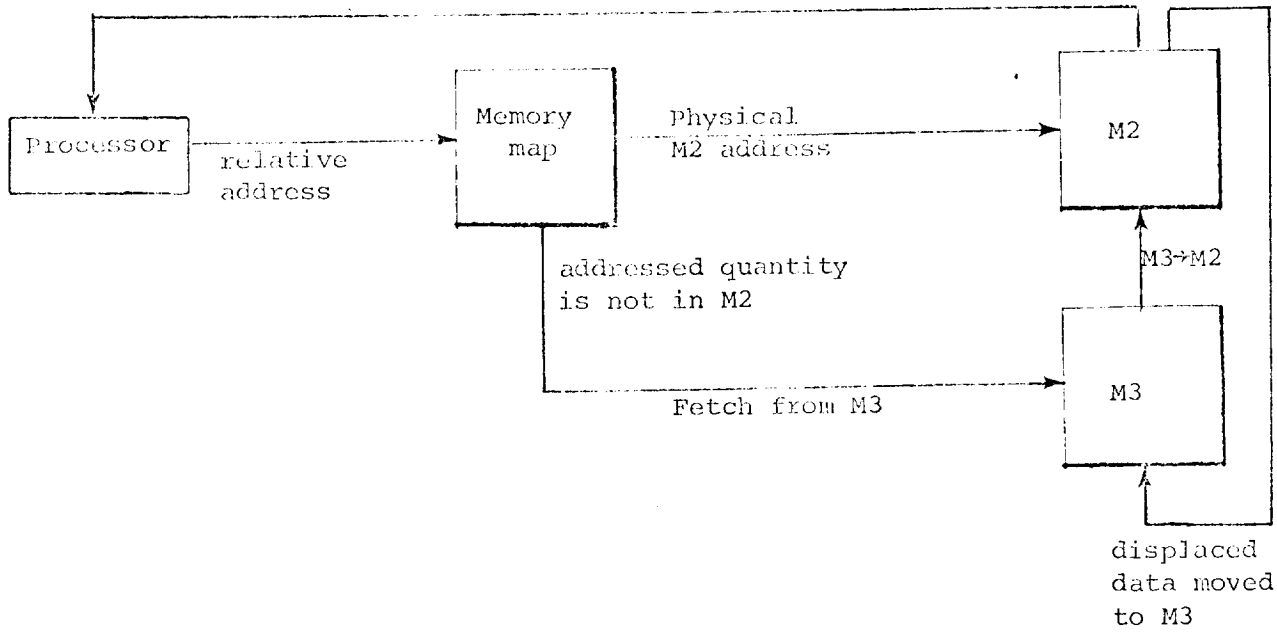
Since a program could be allocated any physical M2 locations, the addresses contained within the program string must be relative and not contain any absolute M2 reference. A translation mechanism must map the relative addresses into absolute M2 address. Many machines employ a concatenation of the address field of the instruction with the contents of some specified base register. Other schemes employ a descriptor mechanism, which is used to provide an indirect reference. In either case, the relative address is first presented to a memory map mechanism which determines if the desired element is in M2 or whether an M3 fetch is required. Figure 4.7 indicates the basic operations involved in memory management.

The memory mapping mechanism usually employs a limited associative memory to contain the most recently referenced addresses. In the 360/67, the contents of the base register is funnelled thru an associative memory. In the Intermetrics' multiprocessor concept the descriptor's address field is translated via an associative memory.

The first suggestion for achieving virtual memory was published by Manchester University in England, in 1962 [7]. Virtual memory has subsequently been implemented in a number of ways, most notably in systems designed to service a large user body generating an unpredictable load and mix of processing jobs (e.g., the HIS 6000 in the MIT Multics system, and the IBM 360/67). The mapping mechanism requires address information to be organized into blocks. Two basic schemes have been defined for handling these blocks:

a) Segmentation organizes address space into a collection of segments which are mapped into variable sized memory blocks

b) paging organizes memory space into "pages" of fixed size.



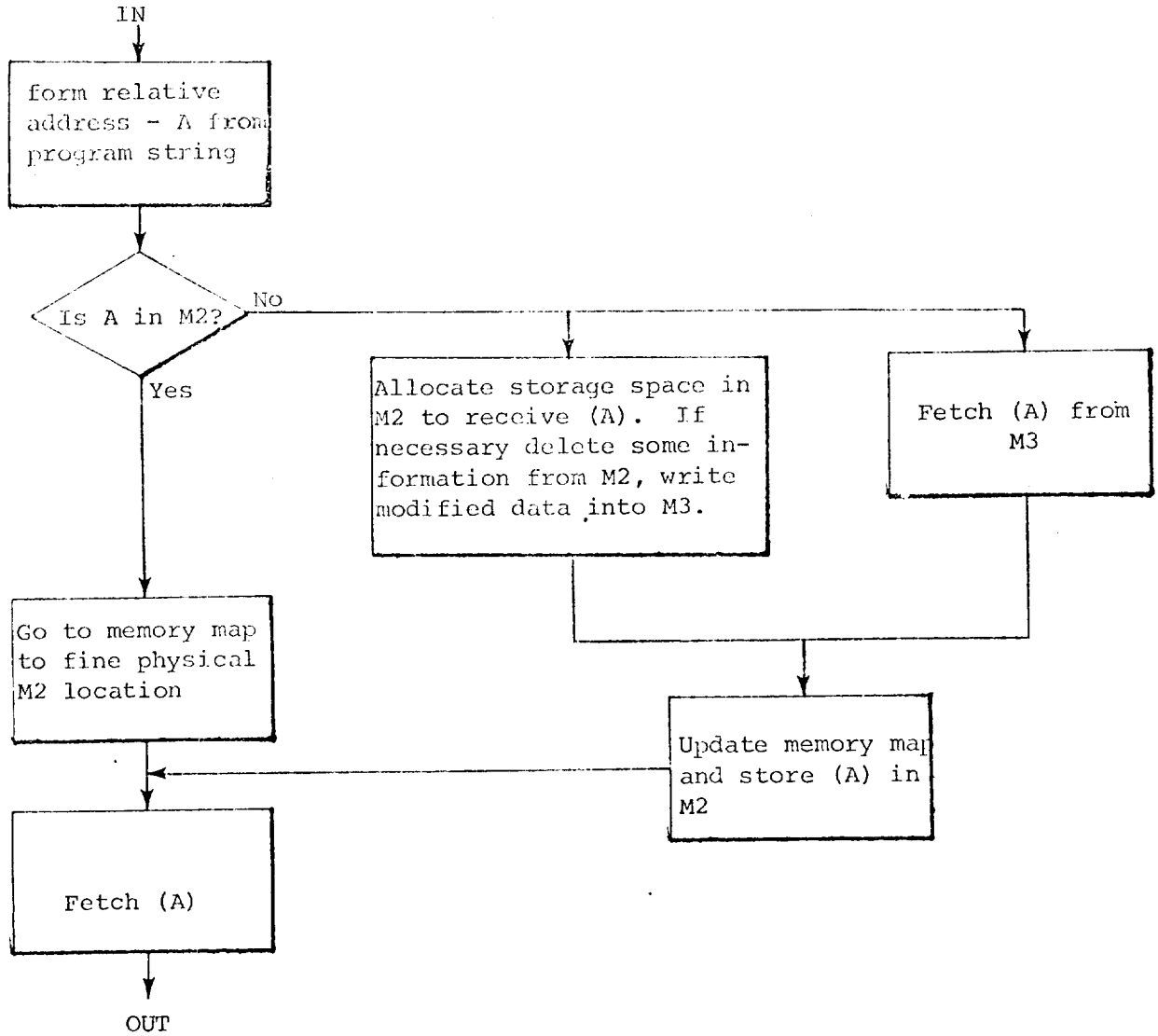


Figure 4.7: Memory Management

4.3.2 Segmentation

Since segmentation is concerned with the modularity and structure of the program it is visible and controllable by the programmer, although usually indirectly through the use of a language. He determines the size of the segment, and attaches its name. Each segment may be considered as an independent virutal memory. Internal to each segment, addressing is relative to the beginning of the segment, and thus becomes independent of addresses in any other segment. This property results in what has been termed two-dimensional addressing: segment number followed by location number. McKeeman [8] points out that this addressing structure is employed in a number of modern programming languages, such as ALGOL, PL/1 and FORTRAN. It is also a property of HAL. These languages use a pair of numbers to represent an address: the first number corresponds to the nesting level (lexical level) of the occurrence of the declaration of the name of the address, and the second indicates the occurrence of the name within that level in the program. The elements of a segmentation implementation mechanism are shown below:

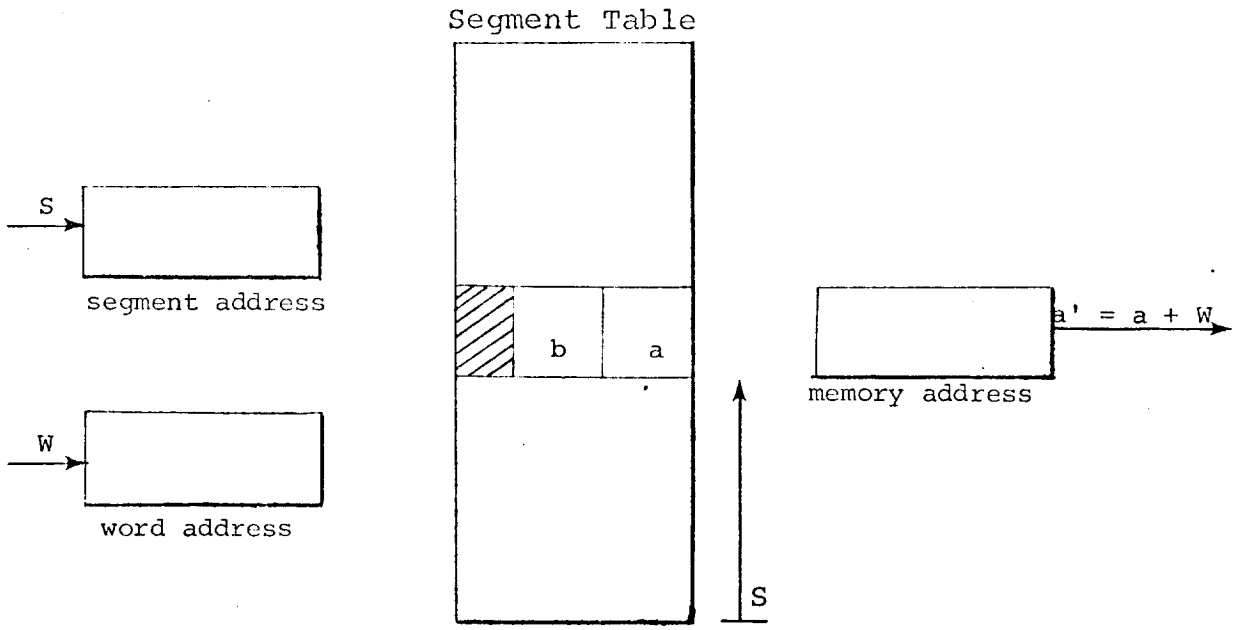


Figure 4.8: Elements of Segmentation

Segments are located by reference to a table, each entry of which is a segment descriptor defining the segment's base address a and its size b. The position of the descriptor in the table is S from the base of the table. A reference to an address in name (address) space is of the form S,W. The component S locates the desired segment's descriptor in the table. If it is not in the table (i.e., the segment itself is not in operating memory) a missing-segment trap occurs. The segment is then brought into operating memory from mass storage, and its descriptor is placed in the table. A test whether W > b is made to check if a programmer's reference is out of bounds of his own segments. Then the location a' in physical memory to which the name space address S,W refers is formed by a' = (a + W). This address translation mechanism can be realized in special hardware, with a set of special associatively addressed registers. Or the tables can be accomodated in operating memory, with all translations performed by multiple levels of indirect addressing. The latter approach involves two or more memory accesses per reference and results in a considerable penalty.

The segmented addressing scheme offers several attractions for a large and diverse software system such as the space station central multiprocessor.

a) Program modularity, Program modules are organized into distinct, separately named and controlled segments.

b) Variable data structures. In a system such as the space station, the data base will contain large and complex data structures which will vary in size and content during use. By creating segments of such structures they may be assigned just the memory they require. Their manipulation is well controlled.

c) Protection. A high degree of access control can be provided by the segmented approach through indirect addressing coupled with access privileges which constrain read and write operations within a given segment.

d) Program sharing. By enabling one physically stored module to be known in different address spaces under different segment names, it may be directly shared between two or more users. This obviates the usual practice of creating copies of multiply used routines, and consequently economizes on memory space.

4.3.3 Paging

Operating memory address space is divided into a number of equal sized pages. Each page is identified by the memory location of its first word. Words within the page are referenced by word number w from the first word. A page is referenced by its position p in the page table. A virtual memory address, a, is equivalent to the pair p,w (in a similar fashion to segment addresses). The total number of active pages may not exceed the page capacity of operating memory. Those pages not being executed are transferred to the next level of storage, thus realizing the concept of virtual memory. Since all pages are equal in size, replacement involves only the problem of finding the necessary equal-sized "holes" in operating memory. "External" fragmentation of memory need not occur.

Page availability is maintained in the page table. The pth entry in the table is the memory location of the page containing address a, where p = integer [a/Z], and Z = page size. If the pth entry is missing, the page does not reside in memory, and must be fetched. This condition is referred to as a missing page trap. If the page is present, the referenced word is the wth element of the page, where w = remainder (a/Z).

Paging is attractive to the system designer as a technique for physical memory allocation, because of the regularity of the equal-sized pages. It is attractive to the programmer because he is relieved of the concern of allocating physical storage, and, indeed, need never exercise any direct control over the mechanism.

A major design decision is the choice of page size. A large page, say over 1000 words, may result in a high proportion of unused page space, if natural program modules are smaller than the page size. This is referred to as "internal" fragmentation. With a small page, less than 10 words, an overhead problem arises due to the large number of pages that must be controlled. The best page size is determined by:

a) Program locality

b) The speed ratio between memory hierarchy levels

Paging cannot achieve some of the advantages of segmentation that were identified previously, because page boundaries bear no natural relationship to program content. Segmentation, on the other hand, lacks the advantages of a fixed size. It requires the availability of contiguous regions of space, of sufficient size to contain the segment. The problem of searching for and/or creating variously sized "holes" in memory is a much more difficult task than matching pages to page spaces.

It is natural to contemplate a combination of the two mechanisms in order to realize both their advantages. The large Multics system at MIT has been the only example of a heavily developed segmented and paged memory management scheme.

4.3.4 Implementing Virtual Memory

When implementing a virtual memory system a number of properties are desired to minimize overhead.

a) An efficient memory map search. This is usually achieved by employing a limited associative memory to hold the most recently used page or segment descriptors.

b) An efficient M2 space allocation algorithm.

c) An efficient determination of the M3 address in the case of a missing page or segment trap. The utilization of a descriptor containing an M2 or M3 address depending upon the state of the presence bit, is convenient.

d) One must attempt to minimize fragmentation of memory into small unusable portions. A memory compaction algorithm might be required.

e) One must minimize the possibility of overloading the system to the extent that thrashing occurs. Thrashing is a state which is reached when memory management begins spending all its time moving pages or segments in and out of M2 and overlaying pages or segments in use. No time is left for processing applications programs. Thrashing can be minimized by providing sufficient M2 and by keeping the unit of memory management small.

Figure 4.9 indicates Intermetrics' approach to memory management via a descriptor-based, stack-oriented structure. Absolute M2 addresses are only contained in "Mom" descriptors. Only 1 "Mom" segment descriptor for a program or data segment may exist. Many "Copy" descriptors may be created with a pointer to the "Mom". This pointer is a two-dimensional address specifying a stack number and offset (SNO). The SNO is the relative address which must be translated into a physical M2 address. The 32 most recently referenced (SNO) addresses are contained in the associative memory. The contents are updated automatically whenever a reference is made. If the SNO reference is found within the associative memory, the "Mom" descriptor which contains the absolute M2 address is retrieved from local storage (M1) and the operating memory address is obtained.

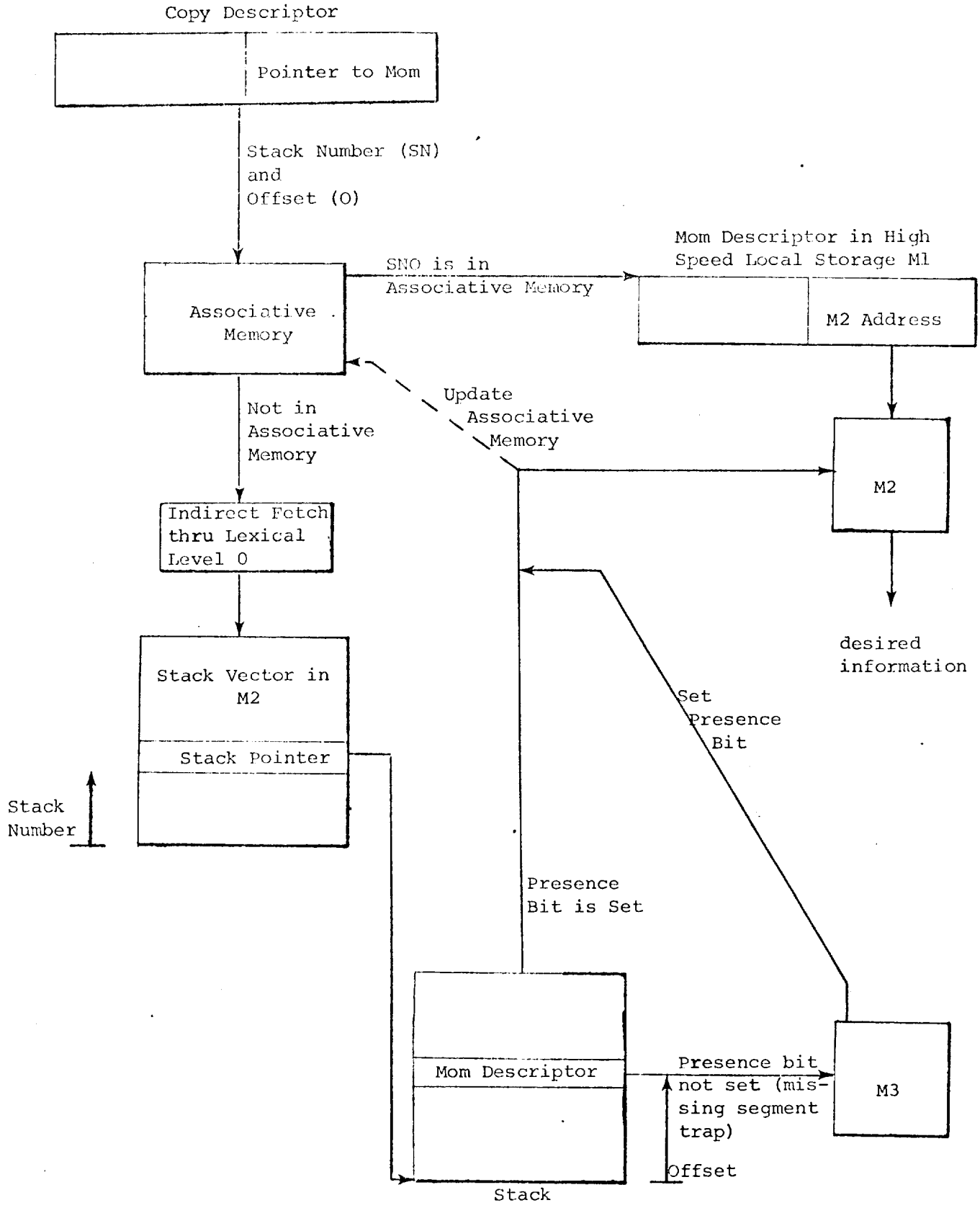


Figure 4.9: Addressing Via Stack Number and Offset

If the associative memory does not contain the referenced SNO, then a three level indirect addressing sequence thru M2 is executed. The first level fetches the stack pointer from within the stack vector using the stack number as the relative address from the base of the stack vector. The second level of indirection is used to fetch the "Mom" descriptor using the offset part of the address as the positioning relative to the base of the stack.

If the referenced segment is not in M2 it must be fetched from M3. This is indicated by a "presence" bit contained in the "Mom" descriptor. If the segment is present within M2 it is referenced directly. In either case the associative memory is udpated so the "Mom" descriptor can be referenced more directly the next time.

References for Chapter 4

1. CRC Standard Mathematical Tables, 19th Edition, p.570.

2. Wilkes, M.V., "Slave Memories and Dynamic Storage Allocation", IEEE Trans. EC-14, April 1965, pp. 270-271.

3. Gibson, D.H., "Considerations in Block-Oriented Systems Design", Proc. SJCC 1967, pp. 75-80.

4. Green, J.P., "Mass Memory Parametric Data", Task Report MD-101, Intermetrics/NAR, June 1971.

5. -----, "Mini-wire Sale Completed", Computer Design, September 1971, p. 12.

6. Denning, P.J., "Virtual Memory", Computing Surveys, September 1970, pp. 153-189.

7. Kilburn, T., et al, "One Level Storage System", IRE Trans. EC-11, April 1962, pp. 223-235.

8. McKeeman, W. M., "Language Directed Computer Design", Proc. FJCC 1967, pp. 413-417.

Chapter 5

ADDRESSING

The question of addressing is the most dominant feature in the diversity of instruction architectures. It can be viewed from many different angles: correspondence to software, ease of usage for programmers, bit minimization, physical implementation and execution, hierarchies of memory, and/or operating system memory resource allocation. We shall discuss several of these aspects and show various options or methods that may be employed.

5.1 Addressing and Instruction Architecture

When an instruction architecture is contemplated several different independent decisions with regard to addressing within an instruction must be reached. The number of operands which an instruction can contain may vary from three, two, or one explicit operand(s) to implied operands, where the implied operands are to be obtained from a stack. The question as to how many hardware registers, of what type, and how they are to be addressed arises (single accumulator or "general" register, hardware "top of stack" for a depth of two, ...). Finally, exactly how is memory to be addressed: all memory addressable, two-dimensional addressing, self-relative, etc.

5.1.1 The Number of Operands in an Instruction

Most operations which occur in algebraic languages are dyadic operators. That is, the operation manipulates two inputs, transforming them into a new output value. It is seen that dyadic operators (+, -, ÷, x, ...) have three operands: two input operands and one output operand. There are, of course, monadic operations such as negate or absolute which have two operands: one input operand and one output operand.

Instruction architectures vary as to the number of explicit memory-addressed operands which appear within the instruction, yet, of course, the necessary three operands for dyadic operators must be present. (Two operands for monadic operators.)

Three memory operand instructions are found in several machines including the Honeywell 800/1800 series. However,

when the actual usage of dyadic operators is examined it is seen that seldom are three different memory addresses needed. Consider for example:

```
A = B;

A = A + 1;

A = - B/C
```

In these, admittedly biased, examples the use of the three memory address operands is wasteful. In the first example, there is but one input and one output. In the second, one of the inputs is also the output address, and in the third a monadic operator appears.

The waste, or non-use, of a memory address is only bad in so far as it takes room. If the instructions are of the three-operand form and not all three operand memory addresses are used, the instruction still must save space for the presence of these memory addresses which are many bits in length. It is, therefore, usually found advantageous to have at least one of the three operand addresses implied.

Two memory operands are occasionally met with in the instruction architecture. In this case one of the two operands besides being an input is usually also the output operand. The IBM 1401 is such an example. This form of two operands can be very useful where most of the operators are monadic such as is commonly found in data processing where much of the computer time is spent in moving data and editing them.

The most common architecture found is based upon single memory address operand instructions. This is common in both the second and current third generation computers such as the IBM 7090, IBM 360 series, Univac 1108, and the DEC PDP-10. With the single memory address operand an accumulator (or another "register") becomes an implied operand for the instruction. Commonly then the implied operand serves both as one input operand and the output operand of a dyadic operator. When monadic operators are used one operand can be the memory address and the other the implied accumulator. When the third generation of computers developed, the "implied" accumulator was often made into a set of general registers of which one could be selected to be the accumulator. This has led to the characterization of the 360 as having a 1.5 operand instruction set.

The single memory address form of instruction is very useful when sequential accumulation of results occurs, such as in:

```
A = B + C + D + E;
```

However, if a tree structure form of computation is needed (as commonly occurs) such as:

```
A = (B + C) * (D + E);
```

the accumulator would have to be saved after calculation of B+C before D+E can be done. One of the hopes of the general registers development in one third generation computer with multiple accumulators was to be better able to do efficient calculations of this form (i.e., save on storage to memory for temporaries).

One of the principal advantages of having fewer memory operands with each instruction is in the space savings to be found by not having useless fields in all instructions. That is, it would be desirable to use instruction space for memory address operands only when they are needed. The ultimate in this form of space savings is to be found in the zero memory address operand instruction. In this case all of the necessary operands for an operator are implied. These are the stack machines where the "top of the stack" provides the necessary number of operands for an operator and the resultant output value is in turn placed upon the stack. The Burroughs B5500 and B6700 are examples of such machines. The memory address operands, of course, must be able to be fetched from memory and stored into memory. These are, in effect, merely two forms of operands.

This stack form of instruction is one of the most efficient ways in which to specify an algorithm since only the minimum amount of information needed for execution need be present.

The stack itself can be considered in several ways. From a HOL point of view the implied operands of the stack correspond to many of the parse algorithms which have been developed for compilation and hence are able to produce extremely efficient code. From a multiple register point of view the stack provides a method for the dynamic assignment of the general registers rather than the static assignment at compilation time with its inherent inefficiencies.

5.1.2 Single Accumulator and General Registers

While many second generation computers had a single accumulator, third generation machines have tended to have a set of general registers. This has come about for several different reasons. Each reason stems from the basic desire for more efficient and quicker execution. As was seen above, a single accumulator does not make for efficient execution of

tree structured statements. Therefore, if several accumulators were available, storing into memory for a temporary could be avoided; this would save both time and space since memory would not have to be referenced. Also technology, by the third generation, had improved to the degree of allowing more complex hardware in the processor. Thus, multiple accumulators could be implemented.

Another aspect is invoked with the addressing of memory. Second generation machines often had separate index registers from the accumulator; these then needed a separate set of instructions for their manipulations and similarly they were then restricted in the operations which could be performed on them (e.g., no multiplications with an index register). The third generation often has truly general registers which can be either accumulators or index registers (or base registers) thus optimizing on the resources of the speedy registers for use as needed.

The desire to use more accumulators was based on the desire to improve the speed of computation by having fewer memory references and by doing manipulations and operations with the general register set. Unfortunately, this very desire forces the instroduction of bookkeeping instructions to set up the registers so that they can be manipulated. It is often difficult to tell from instruction occurrence statistics for an IBM 360 whether the large number of loads (L, LH, IC) used are to keep the register policy happy or are rather a by-product of improvement.

When both base registers and index registers are available their usage is often confused. Base registers are primarily used to address physical locations. They provide the capability of addressing particular regions of core. Their value interpretation is that of a physical memory address. Index registers are used to locate an element within an ordered data structure. They refer to data elements which are to be manipulated and do not inherently indicate physical addressing. If a character array is being indexed, then the elements are in byte units, if word integers are being referred to, the index actually refers to four byte quantities (in the 360). Because this distinction is not maintained the automatic quality of element indexing cannot in general be performed. (In the 360 the SLL instruction proliferates in order to align the "index" properly.)

One other major problem can develop with the use of a set of general registers. This is the question of how to optimally use them. A choice has to be made as to which registers are to be used for accumulator(s), base register(s) or index registers. The static assignment of the use of the registers

unfortunately does not often correspond to the optimum dynamic usage. This is the case since the flow of control through an executing program is simply not known. Alternate paths of execution exist since this is what the execution of an algorithm is about. The use and savings of registers for one branch of an IF...THEN...ELSE statement is, in general, entirely different from that of the other branch. Similarly when a subroutine is entered, the use of registers within the subroutine is not correlated with use in the calling procedure (where the CALL may be issued at distinct locations each with different register usage).

While it could be truly argued that in any case multiple registers are better than one, the actual policy implementation of their usage can cause large inefficiencies.

Above, it was seen that in the zero memory address operand form of instructions a push down stack mechanism is used for operands. This, by its very nature, tends to optimize the usage of the hardware registers available for accumulations and index registers. When a subroutine is entered, the dynamic environment stack continues to push and pop as needed for the subroutine and hence it acts as an automatic dynamic optimization.

When optimizing is tried in code generation by trying to identify common sub-expressions (e.g., I + 1 in $A_{I+1} = B_{I+1}$), the stack can become inefficient. (The code needed both in time and space to save and restore I + 1 is (can be) more than the actual recalculation of I+1.)

5.1.3 How to Address Operating Memory

Various methods of addressing physical memory are found in instruction architectures. All of memory may be addressed, a bank of memory may be addressed, addresses may be relative to the executing instruction or, while only a small portion of memory may be directly addressed, the rest could be addressed "indirectly".

Machines such as the IBM 7094 addressed all of memory. This form of addressing implies that the memory address operand must have the number of bits needed to represent all of memory. Not only is this wasteful, since usually only a small portion of memory is needed in the execution of a program, but it also limits the size of memory which can then be used with the instructions.

In order to both reduce the size of the memory address operand field and to remove the restriction on memory size

(or at least increase the limit beyond foreseeable needs), two dimensional addressing is introduced. This addressing can be in fixed banks where a certain block of memory is in use (in the Apollo Guidance Computer there were four "banks" addressable at any given time: fixed erasable, fixed fixed, banked erasable and banked fixed) and hence memory address operands then refer to addresses within the current block, or a more dynamic form of banking can occur as in the 360 where a base register points to a starting location and a displacement field then refers to an offset from the base.

Thus with the use of 16 bits, 4 bits to indicate base register and 12 bits of displacement, the IBM 360 is able to address up to 24 bits (16 megabytes) of memory. The penalty, of course, is the overhead which must be paid in the setting, using, and maintaining of a base register and the restriction to a maximum displacement of 4K bytes in a program segment without the setting of another base register (or the resetting of the current base register).

Another form of two dimensional addressing appears in those computers which have been designed for the execution of Algol (e.g., B6700). Since the instructions to be executed are reflective of Algol, the data referred to must reflect the name scope restrictions of Algol. The B6700 makes effective use of the name scope restrictions in Algol to have its "base" registers (i.e., Burroughs Display registers) set automatically to the dynamic environment of the addressable data. The B6700 "base" register points to each succeeding lexical level which is addressable within name scope rules. The displacement then refers to a particular entity within the lexical level.

Besides having base registers, as in the 360, which are able to address any region or core, many architectures allow "indirect" addressing. By referring to an address word which is within the area which you can address, you are allowed to "indirect" your reference thru this address word to what it points to. Thus, while only a small portion or memory may be "directly" addressable, all of memory becomes addressable.

It is apparent that when the 360 was designed, the increase to 16 general registers from one accumulator and a few index registers seemed so magnificent that the need for indirect references was deemed not necessary. (The 4π AP-1 which is a flight computer by IBM modified from the 360 instruction set has restored indirection.) It turns out that the use of a few indirect references could save immense overhead on register usage and allocation.

When data is being addressed, the actual number of entities (variable "names", e.g., A,B ... in a program) involved,

in general, is small. This comes from the simple limitation of the human programmer. The amount of storage, however, may be large (e.g., arrays of data, single or multiple dimension). When an element in an array is referred to, an index is used. This phenomena has the very nice property of making the base-displacement form of addressing attractive. While entries can be directly addressed, arrays can be indexed into. The number of different data areas are also generally limited, again due to programming language restrictions and conventions and hence the number of different data regions is in general small and therefore the number of base registers for data addressing is in general not too large.

Instructions have other characteristics. Often a routine will far exceed the 4K byte displacement allowable with IBM 360 addressing from one base register. Addressing of a code segment within a code segment is concerned with control flow and usually has a very local nature. This brings one final form of addressing: self-relative addressing. Often branches occur to simply skip one instruction, or a few as in an IF...THEN...ELSE. By using self-relative addressing for control flow within an instruction stream a very high degree of size compaction can occur; it becomes automatically relocatable without changing any code and the restrictions (e.g., 4K bytes per base register) of the code segment length can be removed.

5.2 The IBM 360 and Burroughs B6700

In order to gain an appreciation of the difference in addressing structures, a comparison between the IBM 360 and the B6700 is given.

5.2.1 Two Dimensional Addressing (Static and Dynamic)

In order to process large computational jobs a large amount of addressable space is needed, but with a second generation machine such as the 7090 all of this space (and hence the limit of the memory size) must be addressable. In this case then, it was necessary to use 15 bits in every operand address. The IBM 360 and B6700 both have two dimensional addressing. The IBM 360 uses a 12 bit displacement which is to be added to one of 15 base registers. This allows for a full 24 bit addressing (of bytes) scheme. Here 24 bits of address space has been compressed into 16 bits of information. The B6700 scheme uses only 14 bits with its operands, where the "base" (DISPLAY) register is defined, with only the number of bits needed to indicate the current lexical level (ℓℓ) (i.e., ℓℓ=1 implies 13 bit displacement, ℓℓ=2 implies 12 bit displacement)

and the B6700 displacements refer to "words". Since program segments in the B6700 are described via a "descriptor", the actual size of memory which could be addressed is only limited by the numbers of bits so used in the descriptor. In point of fact, Burroughs uses 20 bit word addresses in their descriptors.

It is easy to see then that if the memory of a computing system is large compared to the modular size of "programs" (or perhaps even procedures and routines), program string savings are to be found by using a two dimensional address.

There is a great difference, however, between the IBM 360's and B6700's two dimensional addressing schemes. The IBM 360 base registers are assigned "statically" at compile time, and it is up to the compiler to try and optimize base register usage. This optimization is minimal if only one base register is needed within a segment. This becomes difficult in large segments since the dynamic characteristics of the segment modularization must be considered.

This static two dimensional addressing of the IBM 360 has several aspects.

a) By using 4 bits everywhere for base registers the displacement range is reduced, since seldom are that many registers desirable.

b) If a program is "one big" segment, then several base registers are needed and segment boundaries must be carefully watched.

c) If the base registers are set upon entering and upon returning to each module then:

   1) There must be code to do this in the program strings.

   2) Name scope problems arise when variables in a previous level are to be addressed since their base registers are in general no longer in existence.

The B6700 optimizes upon the two dimensional address idea by.

a) using only the number of bits necessary for the current lexical level to indicate the number of bits for the "base register". This leaves the rest of the bits for displacement. (There is also the fortuitous circumstance experienced by all, that the more "inner" a subroutine the "smaller" it is, i.e., it needs less displacement to fully address it.)

b) The base registers point at the beginning of each dynamic module, hence allowing the displacement to reach its most extreme logical dynamic range.

c) Since the usage of the "base" (display) register is unique and well defined, (versus general, e.g., base register, an accumulator or an index register) the initialization and resetting of them can be accomplished automatically. Furthermore, no explicit code in the program string is required and current dynamic name scope is maintained.

5.2.2 Implicit Addressing

Compare the expression:

```
A = B + C;
```

on the B6700 versus the IBM 360:

B6700	IBM 360
VALC C	L R0, A
VALC B	A R0, B
ADD	ST, R0, C
NAMC A	
STOD	

In each case they execute similarly: (fetch C), (add B to this value) and (store value into A). In effect it is the only sequential form possible (i.e., ADD before STORE) for this expression.

However, when temporary locations become necessary a difference appears in the code, although the total effect, must of course, remain the same. Consider A = (B + C) * (D + E):

B6700	IBM 360
VALC C	L R0, C
VALC B	A R0, B

(continued)

B6700	IBM 360
ADD	ST R0, TEMP
VALC E	L R0, E
VALC D	A R0, D
ADD	M R0, TEMP
MULT	ST R0, A
NAMC A	
STOD	

Assuming that there are only a few (in our case exactly one) accumulators being used, during the expression evaluation it becomes necessary to create a temporary.

The creation of a temporary indicates an increase in the program size for two reasons.

a) In general, the use of temporaries is a static decision and hence cannot behave better than the dynamic usage of the stack. Therefore, one needs more "temporary storage" locations than stack storage.

b) But more importantly, in the IBM 360 type of machine, every instruction has an operand, therefore, the temporary requires an address which in turn takes space. The B6700 uses implicit addressing; the needed number of operands coming from the appropriate number of locations on top of the stack.

When temporaries are needed, most often an implicit address scheme allows for the savings of "temporary" operand addresses.

5.2.3 Descriptors

Descriptors can be considered either as sub-operators or as the ideal data structure which is being manipulated. When considered in the first manner, it is seen that the descriptor saves on the program string length. "Fewer" operators need be specified since the "sub" part of the operator is found in the descriptor of the data structure. For example, the IBM 360 has for "add":

while the B6700 has simply "ADD". This of course requires fewer opcodes, and in turn fewer numbers of bit states for the necessary operators.

When the descriptor is regarded as the "data structure", it shows at least two virtues. One is the fact that by being "semantically concise" (further discussed below) it places into one location the complicated description of the data structure, which thereby need not be repeated in multiple references in the program. The other is the observation that the number of entities which are manipulated by a program are few. The reason that a large addressing space is normally necessary is that if the machine does not have descriptors, then each "memory cell" of the data structure must be directly addressable. The example of an array of 100 scalars on the IBM 360 is in fact 100 memory locations. On the B6700 it is one entity: a descriptor which indicates the dimensions of 100 and where it is to be found in physical core. This very important phenomenon reduces the addressing requirements of a program string, since the full physical memory address need only appear in the descriptor. The descriptor becomes one of the "few" entities which must be addressed and hence only a small address field is needed in the program string proper.

5.2.4 Type Differences

Descriptors allow any information which can be "bottle necked" to be placed in the descriptor once, instead of having the information repeated throughout the program string.

Besides having character data (for I/O) and an internal arithmetic form, most machines have in fact several internal forms. The difference between the "character" and "internal arithmetic" comes largely from the savings yielded by compactly storing and manipulating them in the internal form. The various internal forms come from considerations of preciseness.

Types can be optimized by:

a) making one a proper subset of another (e.g., integer is a subset of single precision floating point on the B6700). Thus, the difference between the operators disappears (except for an explicit operator to recover the proper subset; such as INTEGERIZE).

b) the need for multiple forms of the same operator disappears (e.g., IC, LH, L, LD, LE)

c) and the need for explicit type conversion operations is reduced. The program string could be further minimized by providing an explicit operator for each type conversion when needed (e.g., scalar to character, while integers to scalar would be implicit by the integer definition as a subset of scalar).

5.2.5 Semantic Conciseness

Probably the most powerful way to save in program string length is by having semantically compact operators. By having the operator correspond to the operations indicated in the problem language being executed, the minimum amount of translation is needed and hence the minimum amount of expansion in the program string.

The Burroughs B6700 is an "Algol" machine. Its operators are those that ALGOL indicates.

The IBM 360 is semantically concise only to "BAL" which is merely stating a tautology. The IBM 360 is not semantically concise to any real "problem oriented language".

Besides being semantically concise with respect to the operations needed for a problem the operators can be "semantically concise" in the way in which they are constructed. Branching occurs within a program under execution and not logically with respect to all of physical memory. The IBM 360, as most machines, allows the branch address to be any address of physical memory. The B6700 uses relative addressing (that is, relative to the program under execution) either in the same or different segment. This of course reduces the address space necessary, since it corresponds to the dynamic space involved at execution time. The RC4000, although built upon similar concepts as the IBM 360, has relative addressing, and this in turn creates an efficient and small(er) addressing need.

In the IBM 360 each memory reference instruction generally carries 4 bits of indexing information. The B6700 indexes only when needed, and since a stack is used (hence implicit addressing) only an 8 bit operator is needed (which can also load the resultant indicated entity). Assuming that not every memory reference needs to be indexed (the indices themselves must be fetched from memory) the use of indices when needed, (and semantically concise operations make the need less) will, in most every case, minimize the program string length.

The use of short literals also compresses the program string since the constants used are usually small integral

values. Recognition of this fact allows for their representations in the amount of space needed and not the amount for the worst (largest) case possible.

5.3 Implementation Aspects of a Stack Machine

5.3.1 Definitions

The stack provides the mechanism through which implicit addressing can be accomplished in a semantically concise and efficient manner. The control sequencing and addressing with the stack will be discussed in this section. A specific implementation is presented. Details can vary from machine to machine. However, the fundamental ideas will remain the same.

In a sense the stack is a hardware element just as the arithmetic unit is an element. It can execute three primitive commands:

a) PUSH

The PUSH command will take the contents of the stack buffer register and place it on top of the stack. Simultaneously it will shift all other elements of the stack down one level. For example, the old top of the stack becomes the second entry in the stack.

b) POP

The POP command fetches the top of the stack and places it in the stack buffer register. Simultaneously, it shifts the contents of all other elements of the stack up one level. For example, the old second entry of the stack becomes the new top of the stack.

c) Stack Fetch

PUSH and POP store or retrieve information from the top of the stack. In many instances, information is desired from other stack locations. The Stack Fetch sequence accomplishes this function by fetching from the stack location (indicated by the lexical level and displacement) and placing the information in the stack buffer register. Stack Fetch does not change the state of the stack in any way.

One could implement the stack as a word parallel-shift register. This would fix the length and make it a specialized element of the computer. In order to achieve generality and

flexibility in the design, we choose to implement the stack by employing a standard linear memory array with some specialized pointers. These elements are manipulated by micro code to create the three control sequences.

In general, the length of the stack can vary during execution from ten's of words to thousands of words. For this reason the bulk of the stack must, due to practicality, be contained in M2. However, the more dynamic part of the stack (we choose 8 locations) can be placed in M1 for faster access. For the purpose of the following description the stack word size and M2 word size are assumed to be the same.

5.3.2 PUSH

The PUSH sequence, whose flow chart appears in figure 5.1, involves both the M1 and M2 portion of the stack. Figure 5.2 depicts these two portions and provides definitions of the various pointers used to control the stack.

The M1 portion of the stack can be pictured as a wrap-around shift register. The oldest data is pointed to by M1SL (M1 Stack Limit). The first empty location is pointed to by M1TOS (M1 Top of Stack). Whenever M1TOS = M1SL, namely the M1 portion of the stack overflows, the contents of (M1SL) is moved into M2 location indicated by M2TOS (M2 Top of Stack). If M2TOS ever equals M2SL (M2 Stack Limit) then the M2 part of the stack has overflowed and a trap is generated. The stack overflow trap routine could then, depending upon conditions, allocate more storage for stack use and change M2SL.

The data to be entered into the top of stack is contained in M1BR (M1 Buffer Register). Upon entrance to the routine M1TOS is compared with M1SL to see if the M1 portion of the stack has overflowed. If it has not an M1 write is set up. The M1 address is M1TOS and the data is contained in M1BR. Finally, M1TOS is incremented, modulo 8, before the exit.

If the M1 stack overflows a determination is made as to whether the M2 part of the stack will overflow. If so, a trap is entered. If not, an M2 write is set up, using the M2 address indicated by (M2TOS) and the data pointed to by (M1SL). M1SL and M2TOS are incremented, followed by the M1 write set up.

5.3.3 POP

The POP sequence is shown in Figure 5.3. If the M1 part of the stack is empty, an M1 stack underflow condition exists and a read from M2 must be initiated with an M2 address of

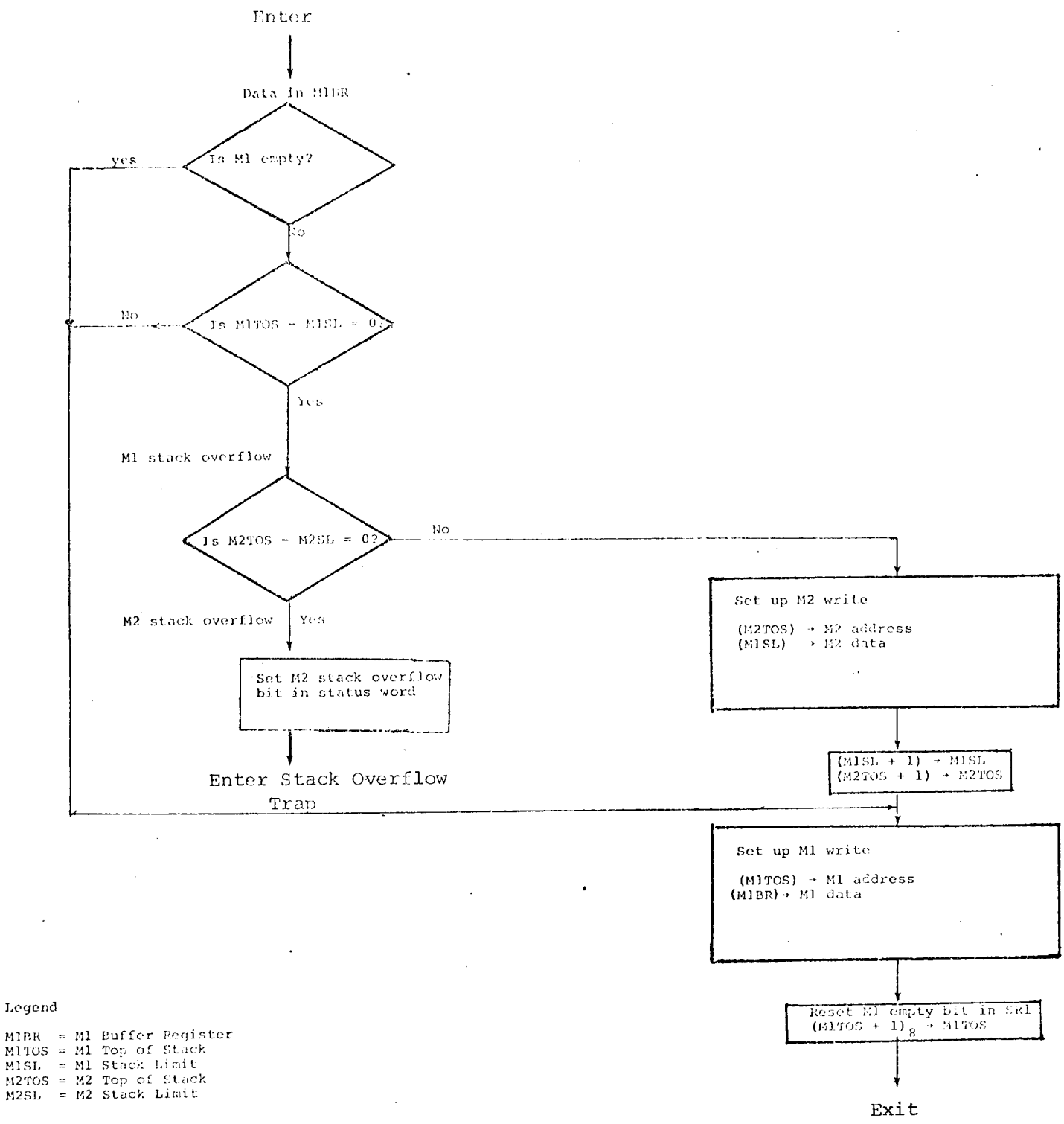


Figure 5.1: PUSH

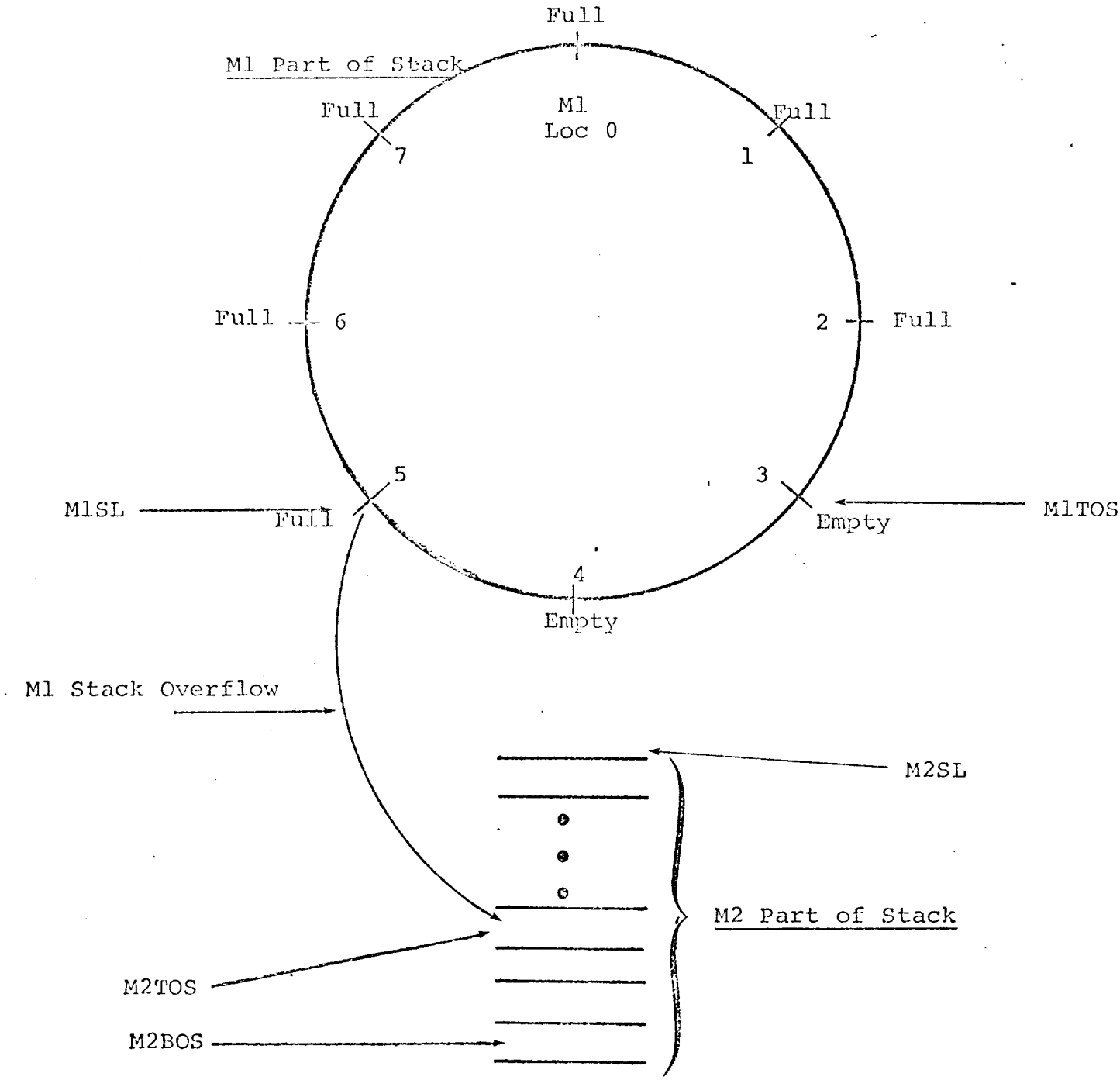


Figure 5.2: The Stack

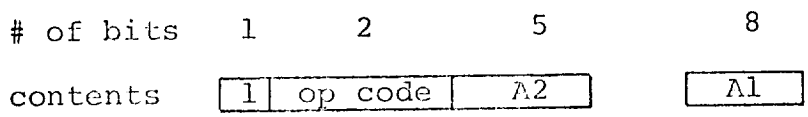
(M2TOS) - 1. The M2 words are placed into M1BR. On the other hand, if the M1 stack is not empty, the contents of (M1TOS) - 1 is read and placed into M1BR.

If the stack becomes empty the condition is set into SR1 for input into the next POP sequence. Every PUSH sequence will reset this empty condition.

5.4 Effective Address Generation (EA). (Lexical Level Offset Addressing)

Within the instruction architecture of a stack oriented machine there exists a class of instructions which refer to information within the stack. Whenever one of these instructions is encountered an effective address (EA) must be calculated. The sequence to be presented depicts a specific design [1]. In general, the details of EA calculation might be different. However, some form of addressing with the stack must be provided.

The format of the class of instruction referencing explicitly the stack is:



The address couple A2||A1 forms a 13 bit field. A_{12}, A_{11}, A_{10}, ..., A_0 which is interpreted as follows:

a) The lexical level indicator, $\ell\ell$, is the key to the interpretation of A2||A1. The first step is to find the positive integer m, where:

$$2^{m-1} < \ell\ell \leq 2^m$$

b) Form Field 1 where

$$\text{Field 1} = A_{12}, \ldots, A_{13-m}$$

c) Fetch from M1 the base register specified by Field 1. Denote this base register by BRm.

d) BRm is in Stack Number, Offset representation.

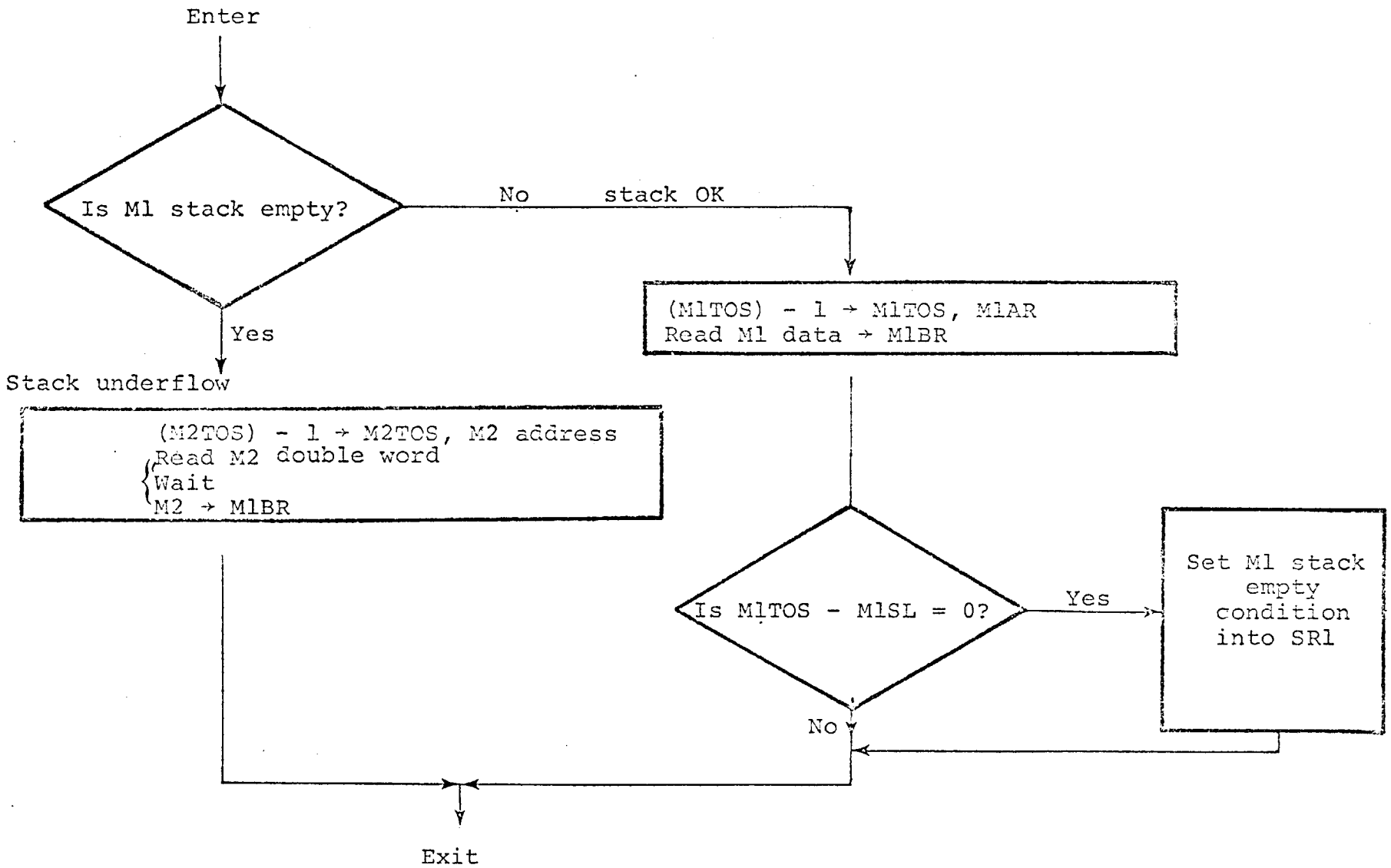


Figure 5.3: POP

e) Next Field 2 is formed

$$\text{Field 2} = A_{12-m}, A_{11-m}, \cdots A_0$$

f) Finally the effective address (EA) is formed where

$$EA = (BRm) + \text{Field 2}$$

This addition only occurs to the offset portion of (BRm).

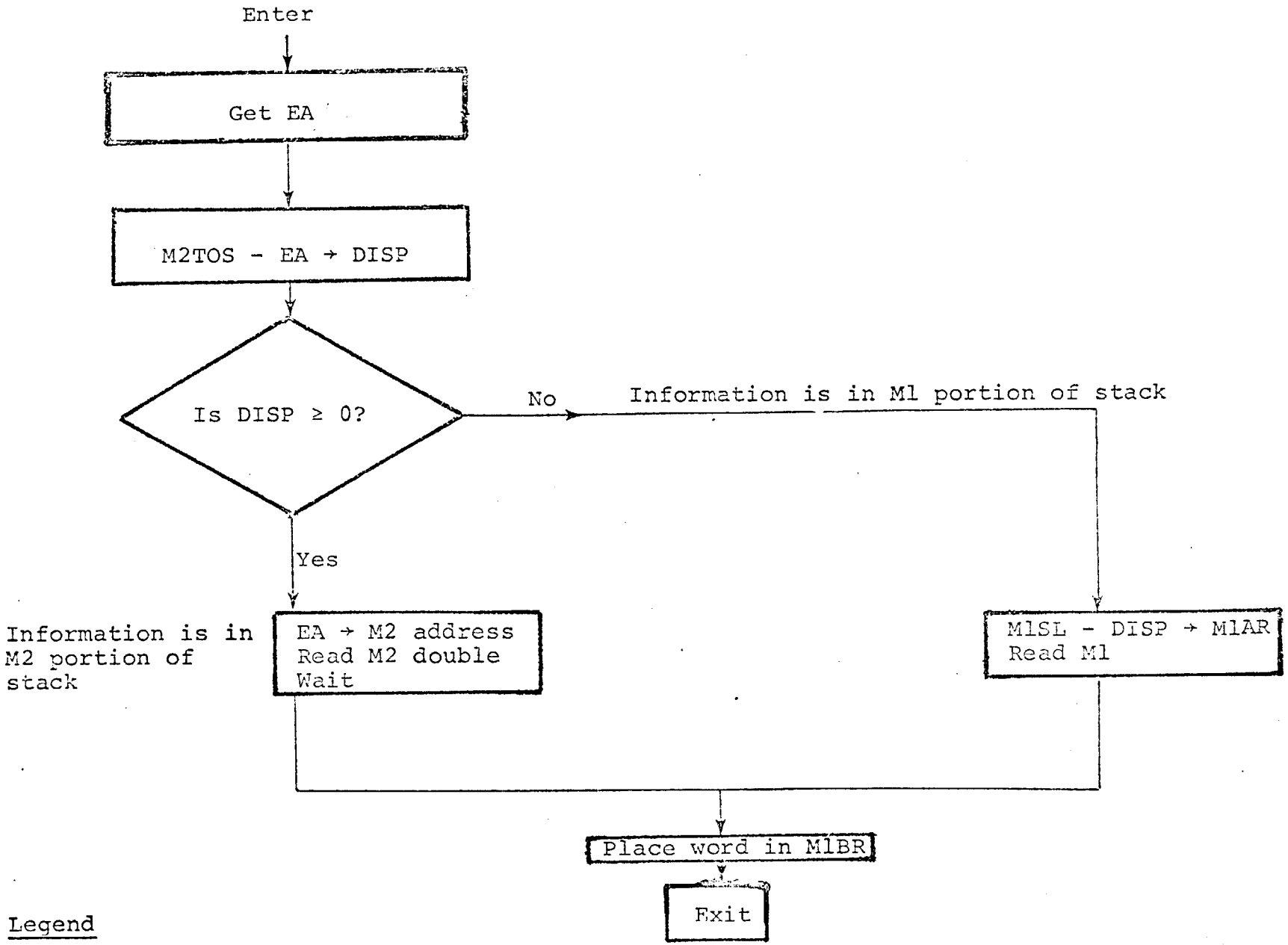
5.5 Stack Fetch

When information is required from any location except the top of stack, a stack fetch sequence must be executed (see Figure 5.4).

The main test to be performed is to determine whether the information to be fetched is in the M1 or M2 part of the stack. This is accomplished by the calculation of the displacement DISP. Information is then read from either M1 or M2 and placed in the M1BR.

Reference for Chapter 5

1) Intermetrics, Inc., "Final Report -- Engineering Study for the Functional Design of a Multiprocessor System", Prepared Under Contract NAS9-11745, September 1972.



Legend

EA = effective address
M1AR = M1 address register
M1BR = M1 buffer register

Figure 5.4: Stack Fetch

Chapter 6

I/O CONSIDERATIONS

6.1 Space Station System Requirements

The I/O interface of the computer which serves the central computational and control element of the manned space station is likely to be characterized by the following observations:

a) There will be a large number and variety of interfaces with diverse avionics equipments. The recent Phase B Space Station analysis has advocated the use of a time-shared, high speed (10 MHz) avionics data bus to simplify the problem of meeting this requirement. The I/O implication of such a data bus will be discussed in this report.

b) The computational speed and storage capacity requirements of the Space Station are such as to make the multiplexing of operating memory an attractive economical proposition. (The cost of storing one bit in a core or plated wire memory is over one thousand times the cost of storing it on a disk.) Until the more exotic, non-moving media, secondary storage technologies (such as magnetic bubbles) become fully operational, the more conventional magnetic drum and disk will probably provide the mass storage capability on the early space stations. The relatively long access time of these devices has made it necessary to treat the problem of getting information in and out of them as an off-line task in parallel with the main computational functions. This chapter will discuss the use of a drum or disk as the tertiary level of a memory hierarchy and as the primary storage for files.

c) Although the Space Station central multiprocessor will possess the powers of a typical large ground based computer facility, it is not anticipated that its work load will encompass as wide a variety of jobs, languages, or users. Perhaps of even more importance, the work load will be much more predictable. This is certainly true of the operational requirements, and even the eventual experimerntal support function will probably be farily carefully tailored to the available facility. The

implication of this for the I/O function is that there is less need for a highly generalized interface to a wide variety of the conventional peripheral equipments, and much less need for the sophisticated data management facility usually found in the I/O hardware and software for controlling these peripherals and providing for the orderly management of a large number of files. It will be assumed that the only need for standard peripheral I/O channels in the planned SUMC MP will be to satisfy the needs of a laboratory environment (e.g., card reader, line printer, operators' console), and that the eventual operational I/O will be performed almost entirely through the avionics data bus.

d) The emphasis on the generation, processing and recording of large amounts of data from experiments places the high density, high speed tape store into a special category of space station I/O device. Even if an improved bulk storage technology is eventually employed in this function, the need for transferring and retrieving large blocks of data from archival storage at rates on the order of several million bits per second will still have to be met. This data originates at the experiment sensors, and enters the system for processing and reduction via the main data bus, which, as will be seen, can typically supply 2.5 million information bits per second. It is felt that a more specialized interface than just another port on the bus is required for this I/O function.

The major impacts of these observations on the I/O hardware and software will now be discussed.

6.2 Data Bus I/O

In order to make more than sweeping generalizations, some assumption of data bus characteristics must be made. Studies to date [1] have shown that an initial Earth Orbital Space Station can be serviced by a data bus whose elements are shown in Figure 6.1, and which has the following typical characteristics.

Multiplexing	TDM
Frequency	10 MHz
Number of devices (stations)	256
Command structure	Command/response

These are the important control characteristics from the point of view of I/O communication.

Command/response implies central computer control. Bus I/O takes place only on the behest of the computer; no device

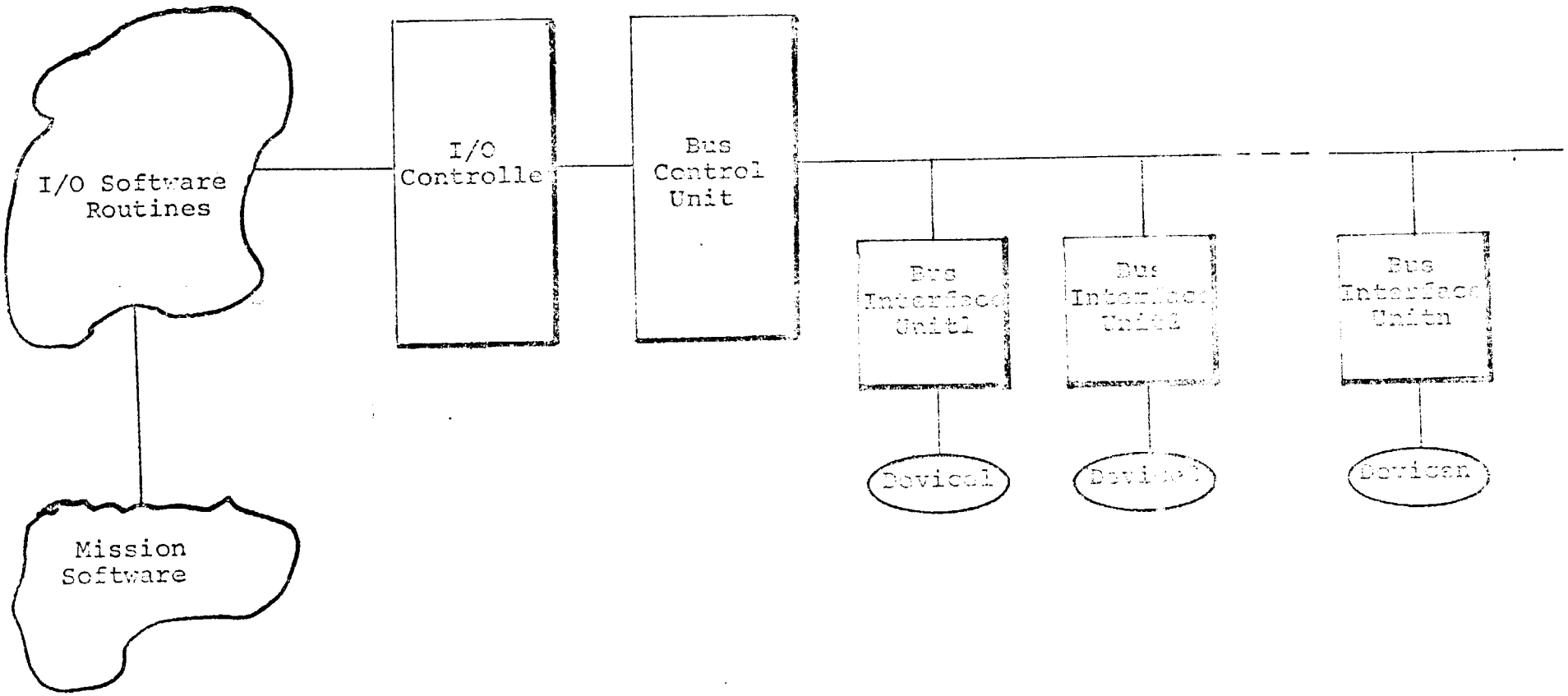


Figure 6.1: Basic I/O/Data Bus Configuration

may volunteer information. It is our opinion, however, that although a strict C/R control policy may be shown to be quite adequate at this stage of Space Station development, it will be advantageous to provide a bus interrupt capability. This is not so much in order to provide the devices with control authority, but rather it is in order to allow the bus control unit (BCU) the ability to off-load the computer I/O routines of chores such as error monitoring, detection of unusual conditions, response to unsolicited communication from Station subsystems, etc.

Local processing at the device level has been proposed to off-load from the bus any high speed repetitive functions (such as strapdown inertial system algorithm evaluation). It is expected that bus communication between computer and device will be composed of short blocks of data from one to several bytes in length, typically 1 to 128. Data transfers of larger blocks (e.g., CRT display frames, experimental data recording) are usually not time critical, and may be achieved by repeated bus I/O. If 8 bytes suffice for device address and address echo check, and assuming a typical 80:20 mix of short (4 byte) and long (128 byte) bus communications, the time to service 256 devices is derived below:

	Bytes			Bits			
	Control		Data	Total/Device		X(# of devices)	Total
	Command	Echo					
Short	4	4	4	12	96	206	2.10^4
Long	4	4	128	136	1088	50	5.10^4
All messages							7.10^4

A complete service cycle of all devices on a maximally configured bus thus generates 70K bits. For a 10 MHz transmission frequency this cycle can be repeated every 7 milliseconds. In practice, delays due to finite transmission speeds will increase the cycle time, but a 10 ms to 20 ms bus service cycle seems to be entirely achievable. A 20 ms cycle, with the preponderance of long communications assumed, will generate about 300K bytes/sec of actual data, i.e., a data rate comparable to that of the higher speed storage devices such as drums, disks, and tapes. However, a data bus differs significantly in the manner in which this data is addressed and controlled.

The type of bus described is essentially a table-driven device: in practice, communication between the computer and the avionics devices will occur as follows:

a) A number of device interfaces will need to be accessed for real time data at the highest service cycle frequency, i.e., every 10 ms to 20 ms.

b) Others will require accessing periodically, but at lower frequencies than the maximum.

c) Some will require occasional sampling of random intervals.

d) Some devices may be attached but may not be components of a computer activity. Nevertheless, their status and health must be continuously known.

e) The remaining interfaces may not even be attached.

The mix of devices in each category is a function of mission phase and/or station operations. It is a delicate design problem to ensure that all the highest frequency requests are complete without exceeding the basic bus service cycle, and without losing some of the less frequent requests. Since these constraints are known only to the system implementer, specific bus configuration should not be wired-in to the hardware (or system software) of the computer or I/O controller.

The device accesses can be organized into a set of I/O tables. Each table contains the list of accesses to be accomplished at a given frequency. Figure 6.2 illustrates an example of such a table, made up of entries for bus I/ O to be accomplished for K = 1 (every service cycle), K = 2 (every other cycle), K = 4 (every fourth cycle), and so on up to K = 64. K need not be in powers of two, but it is felt that this makes table mechanization much easier, and is not a serious burden to the avionics system implementer.

Each entry in the table is a request for bus I/O. Such a request may consist of one or more words with fields which contain the following information:

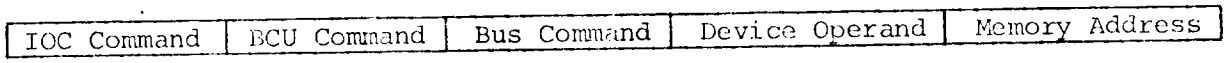


Figure 6.3: Typical Bus I/O Request

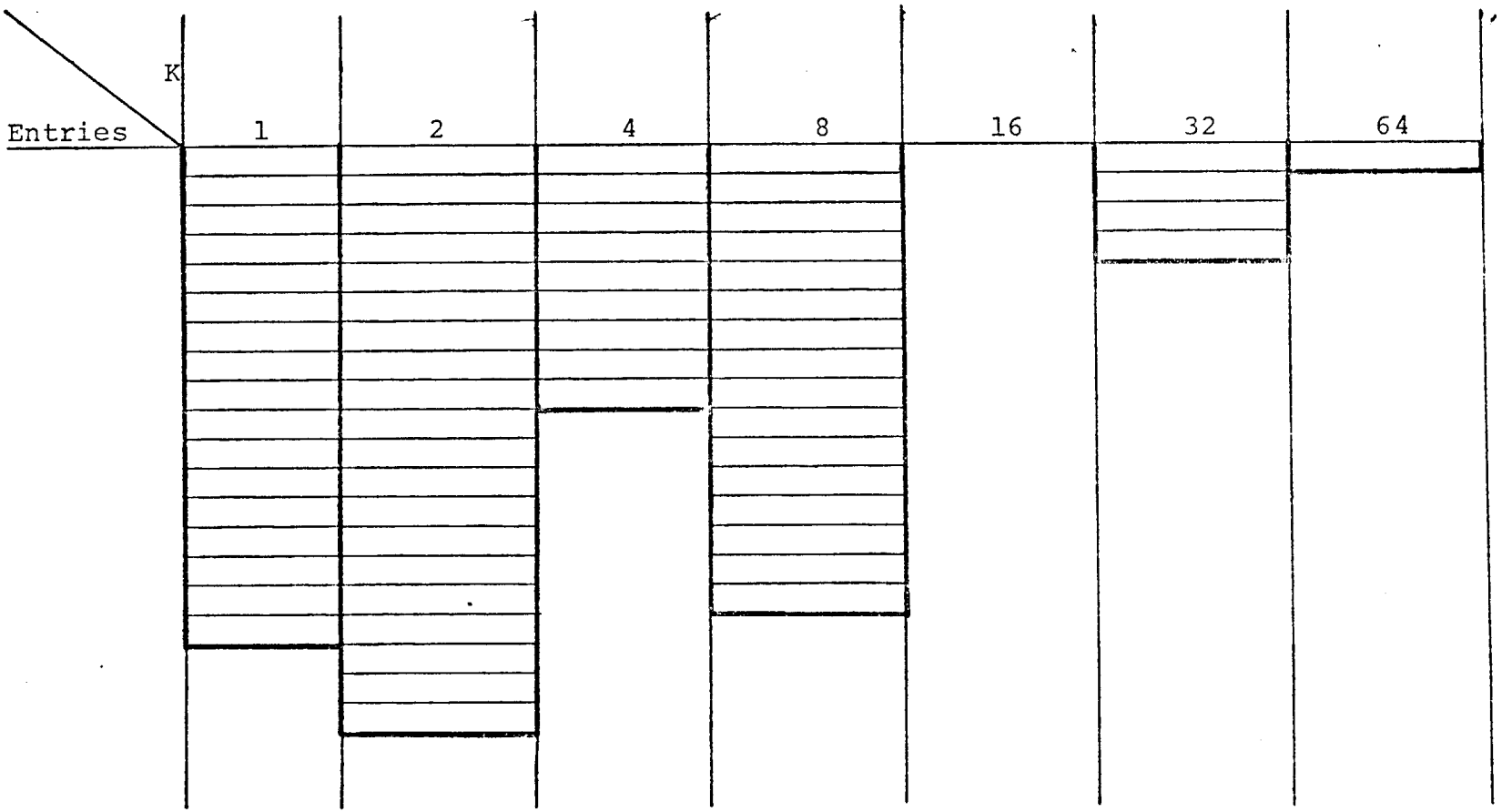


Figure 6.2: List of Bus I/O Requests

a) I/O controller field specifying I/O channel type (i.e., bus) channel number, channel command.

b) Bus controller field specifying special instruction to BCU (e.g., table update, check device status, etc.)

c) Bus command field specifying device address and bus operation (e.g., read, write, set mode, get status, etc.)

d) Device operand field specifying operation to be performed by specific avionics subsystem (interpretation known only to device)

e) Destination field - address and length of memory area in which result of bus I/O is to be placed, or from which output is to be taken.

As each I/O request is executed the appropriate data is transferred between memory device. The question now arises: how is the table of I/O requests to be interpreted and where does it reside? Several alternatives present themselves:

a) It resides entirely in main (operating) memory and each entry is treated as a separate I/O request to the software executive I/O routines. If there is a large number of high frequency entries this will create an I/O bound condition, and much process swapping in a multiprogrammed environment.

b) The table of I/O requests resides in the I/O controller and is executed there independent of main processing. Only the result of each request is transferred to memory. This relieves the interface between the I/O controller and the operating memory of traffic generated by control statements.

c) The table of I/O requests and the resulting data reside in buffer storage local to the I/O controller. Data transfer is in block updates between minor cycles.

The progression from a) to c) implies an increasingly elaborate I/O controller. It also incurs the problem of buffering the bus I/O data. If a user program no longer has the ability to place each individual request, than it has no knowledge of when an update to (or from) the requested bus device is made. This is especially critical for blocked data, where it is essential to ensure homogeneously updated elements of the block. A mechanism for preventing multiple access to data blocks must be provided such as a TEST and SET operator, or multiple

buffers with switchable pointers: the first incurs delays (critical to an I/O process), and the second consumes memory space.

The localization of bus I/O in the IOC allows high frequency bus-computer communication to be conducted without the several milliseconds delay normally associated with I/O devices such as drums or disks, and obviates the need for process swapping to maintain throughput. The low frequency or random bus communication can be handled in a conventional fashion as a single I/O event. Such requests can be treated as temporary insertions into the bus I/O request tables, which are removed when serviced by the bus. Completion of the request can be signalled by an I/O complete interrupt. Division of bus I/O requests into repetitive and random categories depends on the trade-off between IOC complexity, I/O buffer size, bus service frequency, and throughput.

6.3 Mass Storage I/O

The most critical function of secondary storage is as part of the multiplexed operating memory hierarchy. Whether the technique employed organizes memory into fixed size blocks (pages) or variable sized blocks (segments), it is essential to be able to locate and transfer to and from secondary storage fairly large amounts of stored information (from tens to thousnads of words), in a minimal time.

The traditional disk or drum memory systems possess characteristically long latency and/or access times (on the order of tens of milliseconds), and data transfer to those devices is performed in parallel with other CPU activity by an independent processor. It is anticipated that early Space Stations will still employ rotating magnetic storage devices and that I/O will continue to be concerned with their optimal usage. It is important to realize that a subsequent change to solid state mass storage (with little or no access delay) can radically modify the concept of memory multiplexing, to the point where it may not be done via the I/O controller. In the present discussion, we will assume the conventional core to disk interface requirement.

The major concerns with optimal usage of the disk are:

a) Since access times are long (typically 10 to 100 milliseconds), but transfer rates are high (typically 5 to 10 MB/s), it is desirable when a request for a missing memory block is honored, that as much "useful" associated information is transferred along with the specified block, since the cost of so doing is relatively

low. This involves maximizing the "locality" of the executing program which creates the I/O request, or otherwise anticipating its accessing behavior.

b) Since requests for data take so long to honor, it is probable that, by the time a requested block is located and transferred, the requesting process is probably no longer running. It becomes desirable to allow the memory management to determine, at its convenience, when to alert waiting processes of their complete I/O requests. This may be done by causing a table of completed I/O requests rather than to signal the system via an "I/O complete" interrupt, as is usually done. This may be done by causing a table of completed I/O requests to be accumulated by the I/O controller, and only when no further requests are pending, cause the I/O controller to interrupt the system to notify it that all requests have been expedited. A "quiet" I/O complete scheme such as this is expected to greatly minimize the "thrashing" of memory transfers that occurs when operating memory becomes overcommitted.

c) The assignment of disk space can become as critical as that of operating memory. For a high degree of memory multiplexing, disk space can become badly "fragmented" with use, necessitating a compacting or rearranging of the assignment of files. In a real-time system it may require prohibitively long search cycles to update all references to files that are re-assigned. Disk addresses can be organized in a central directory which maps logical into physical address space. This can be accomplished in main memory, at considerable cost of space, or on the disk, at the cost of more complex hardware in the disk controller.

d) Other traditional I/O problems (such as the trade-off between I/O request frequency and I/O buffer space in main memory, and the related question of logical file blocks and how to assign them to a device that is organized into physical records) still remain in a Space Station environment. But, as stated in the beginning, these questions are of less significance in an environment whose work load and user requirements are less variable and more known. A less generalized approach to file directory management may be possible than is found in general purpose ground-based facilities such as the larger IBM 360 installations.

6.4 I/O Controller Design

This section will describe the functional elements of a proposed I/O controller design. Detailed implementation questions are beyond the scope of the present contract. Figure 6.4 indicates the basic functional elements.

6.4.1 Central Control (CC)

The central control unit provides the decoding of the I/O operations, for the initiation and synchronization of commands, and for data transfers between the units. The CC contains an arithmetic unit and the logic required to perform conditional decisions. The sequences issued by CC are stored in a micro control memory and are initiated via commands from the various interfaces.

6.4.2 Interprocessor Communication Interface (IPCI)

Some mechanism is clearly required for communicating between processors and the I/O controller. This is necessary for interprocessor interrupts, I/O commands, and recovery form processor faults.

The IPCI provides the interface to the interprocessor communications bus. One may reasonably question whether a separate interprocessor communication interface is required. Can not all the communications go through M2?

If all the interprocessor communications occur by writing into M2 and reading from it, then the answer to the above question is no! The overhead due to constantly polling M2 would waste processor time and create excessive M2 contention.

If processor communication uses the internal bus, as a communications media, by-passing M2, then the answer is probably yes. The use of the internal bus as the communications media is just an implementation decision. The fact remains that distinct communication between processors and between processor and I/O must occur, outside of M2. The logical decisions performed by IPCI must exist whether a physically separate interprocessor communications bus (IPCB) is employed or not.

A wide variety of signals are communicated over an IPCB. Some are between processors. Others involve I/O transactions. Some examples are given below:

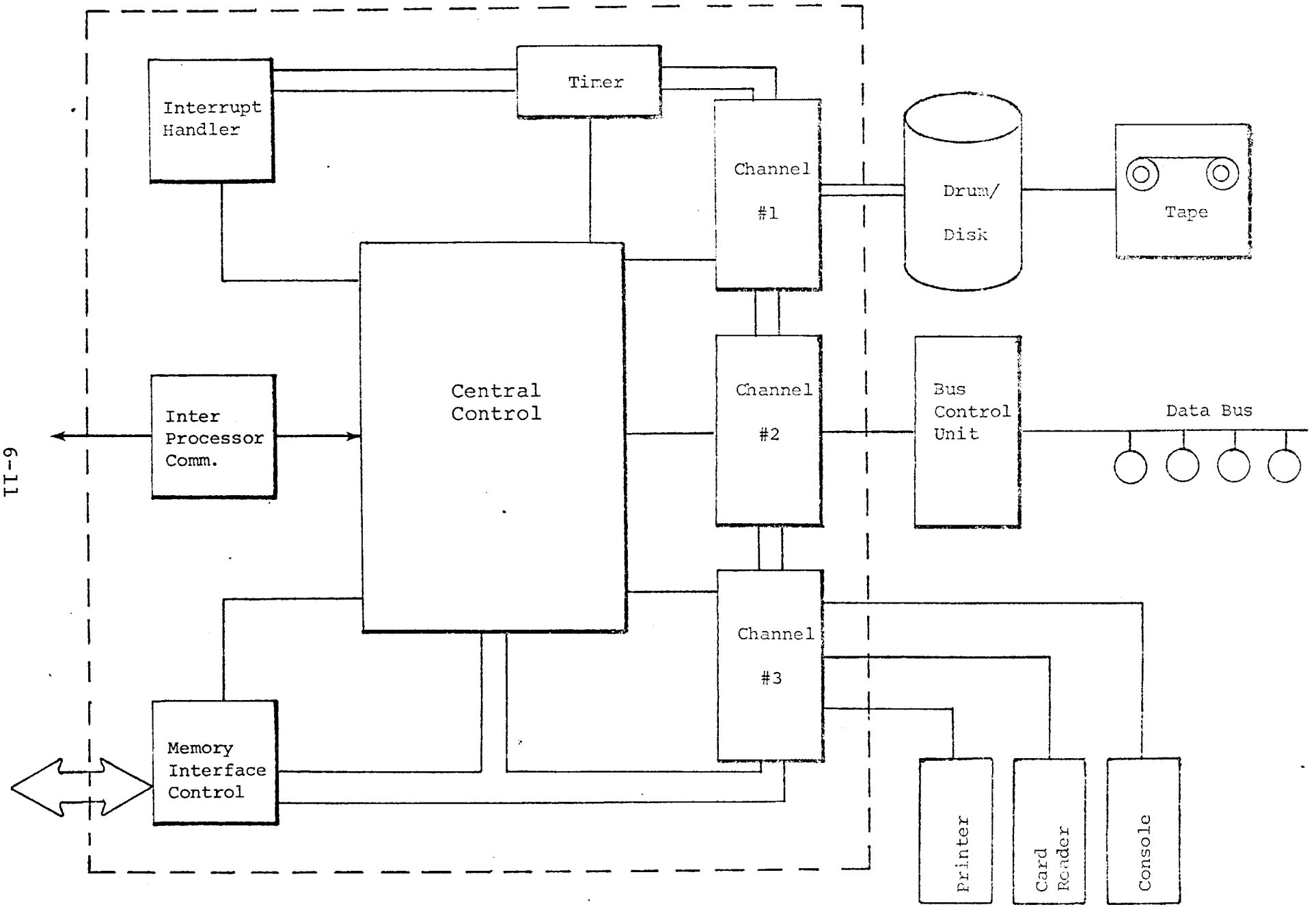


Figure 6.4: Elements of I/O Controller

a) If local memory M1 is employed, then a potential problem exists in updating common information (for example, descriptors(contained within different M1's. The control of the updating requires interprocessor communications.

b) The loading (initialization) and dumping (for a process swap) of M1 can be triggered within a processor or commanded from another processor (in case of an error condition).

c) When a processor fails or detects an M2 failure this information must be signalled to another processor.

d) All the commands issued by I/O executive routines must be sent to the I/O controller over some communicating link.

e) All "done" or "error" interrupts generated by or passed on by the IOC must be steered to a processor over a communications link.

6.4.3 Operating Memory Interface

This interface element controls access to memory by the various channels. It is, in effect, the DMA channel for the I/O controller. The priority as to which I/O interface has access when contention exists is fixed. The following is suggested:

Priority 1 (highest) Channel 1: The devices which operate in the burst mode must be serviced at a rate consistent with their data rate. M3 can possess a data rate of up to 10 MBPS, which is three to six times less than the M2 data rate. However, channel 1 devices cannot sustain a large delay between a request for an M2 transfer and the final servicing of the request since the addressed record is usually not fully buffered and M2 and the auxilliary device must be synchronized during a data transfer.

Priority 2 Channel 2: The devices which are driven by tables in the local memory of channel 2 present to M2 a data rate three to six times less than that of channel 1. Yet, if too much delay is introduced in each M2 transfer, the minor and major cycle times might be exceeded.

Priority 3 Central Control: When the CC receives a command over the IPCB it often has to fetch an I/O control word from M2. While this fetch can be delayed a reasonable amount of time, queueing of too many IPC commands before execution must be avoided.

Priority 4 Channel 3: The devices attached to channel 3 are all slow speed and involve only a few bytes per transaction. A delay of ten to even one hundred M2 cycles will not appreciably affect the performance of these devices.

Priority 5 (lowest): Since the interrupt priority and timer elements of the I/O unit do not use M2 to a significant extent, these elements are placed in the lowest priority category.

6.4.4 Channels

These control the interface to the device categories defined previously, namely:

a) the high speed disk (or drum) and tape

b) the avionics data bus

c) slow speed unit record equipment.

Each channel will contain buffer capacity appropriate to the device, and a set of instructions tailored to the control requirements of the device.

6.4.5 Interrupt Handler

Although not a unique location for the interrupt control mechanism, the I/O controller often contains this function. There is some advantage in handling external interrupts and processor traps with the same mechanism.

6.4.6 Timer

The real time aspects of the MP system require access to a precise time standard. Also the capability of generating an interrupt at a predetermined time, probably by means of a count-down mechanism, is required. Each counter must be addressable from a processor for initialization or readout. These counters are placed inside the I/OC for convenience, thus saving the cost of providing a unique piece of equipment.

6.5 I/O Configuration Organized for Recovery

The I/O configuration presented in Section 6.4 indicates that a single I/OC is capable of servicing the multiprocessor. If this design approach is taken, how can this single I/O meet the requirements dealing with recovery from a failure?

If two or more I/O units are required for system operation then the recovery aspects of the I/O can be made very similar to those of a processing unit. Each of the I/O units would be configured like a processing unit with an M2 interface, a special interface to the Processors via dual redundant communication links, an M3 interface, and a data bus to the outside world. Single instruction Restart could be employed as the major recovery mechanism.

Since only a single I/O unit is proposed to meet the performance requirements, a triple-redundant I/O unit with voting logic is a candidate design approach. Many transients are completely masked in this configuration. If a permanent failure occurs then the voting elements can be reconfigured to comparators and the bad I/O unit taken off line for repair.

Figure 6.5 shows a possible redundant I/OC employing the components described in Figure 6.4. The major features of this configuration are described below.

a) The triple redundant I/O hard core contains the central control, timers and the interrupt control. A failure in this critical area will allow the system to keep running without propagating the error.

b) In order to interface the TMR section with other dual redundant interfaces, voters and switches are provided. The S elements, which are controlled by their associated I/O elements, are used to select which of the dual redundant interfaces to accept data from. The V elements vote upon the triple redundant I/O outputs and produce dual redundant outputs. The voters will automatically reconfigure to comparators and switch out a faulty I/O where required.

c) The IPCB, M2, M3, and data bus are all postulated to be dual redundant. For this reason their interfaces are shown to be dual and they interface to the I/O via the S's and V's. The multiplexer channel which contains peripherals necessary to operate a laboratory model is only shown as a simplex subsystem, with a corresponding single interface.

d) It is assumed that all the peripheral devices attached to the data buses and the M3 controller possess characteristics which will aid in the recovery process. These characteristics include:

    1) hardware to aid in fault isolation between dual redundant threads

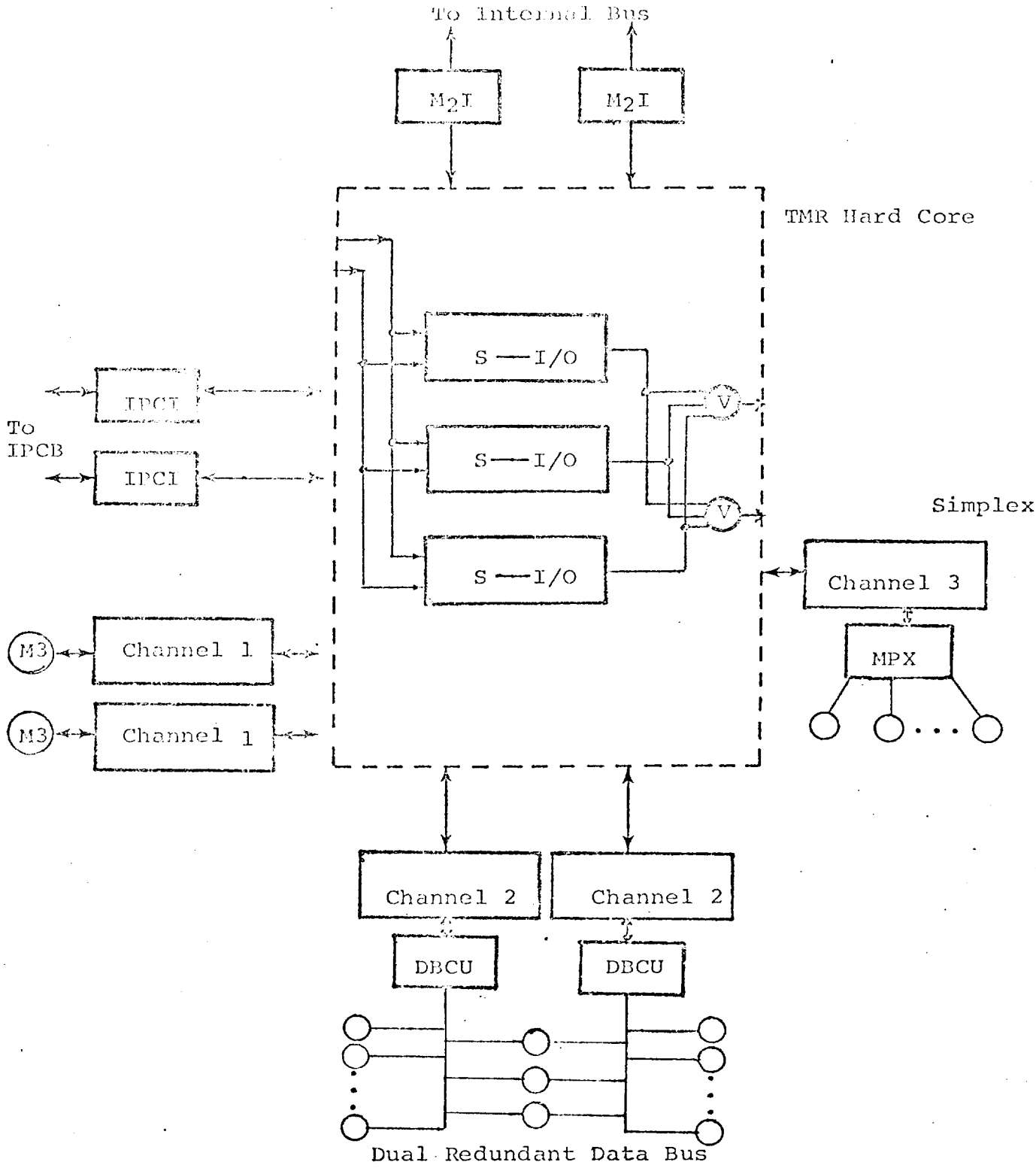


Figure 6.5: Redundant I/O Configuration

2) sufficient buffering, so that aborted commands cannot hang up a subsystem

3) the ability to be reset and to indicate upon request the status of the I/O device

e) Certain problems caused by locking of processes to I/O devices must be resolved by the operating system. This requires the capability of selectively deleting the I/O command created by a process which is cancelled (either purposely or as the result of a failure) from the appropriate device queue. Also, the capability of relieving any M2 space allocated as the I/O buffer area must be provided.

One of the main motivations for a triple redundant I/O central core is to reduce this problem as far as I/O failures are concerned. A failure within the central TMR I/O cannot propagate past the voters. However, a voter or channel failure can cause a temporary suspension of I/O or a re-issuing of an I/O command and the associated problem of releasing any I/O locks.

References for Chapter 6

1) North American Rockwell, Space Division, "Modular Space Station Phase B Extension - Information Management Advanced Development Report", Contract NAS9-9953, MSC-02471, July 1972.

Chapter 7

FAULT TOLERANCE PHILOSOPHY FOR THE SUMC MULTIPROCESSOR

The purpose of this chapter is to present the study results in terms of error detection, fault isolation and recovery philosophy as applied to a multiprocessor system.

7.1 Requirements

The requirements postulated for the system, as a result of the study, are delineated below.

a) The only interaction that the applications programmer should possess with the fault tolerant aspects of the system is to specify whether and under what conditions a program or sequence of events is to be critical. A critical program is defined to be one which must be recoverable in the event of a fault. A non-critical program is one which need not recover.

By classifying a program as non-critical certain design considerations must be kept in mind. The abrupt termination of a non-critical program in the middle of any instruction should not create a situation which will prevent the execution of other critical tasks. Any Compool data which is used by a non-critical program can not be left locked. The failure of a non-critical program can not lock out a piece of peripheral equipment from use by a critical program.

b) It seems reasonable that for certain applications a recovery time of 10 to 100 ms could be required, especially for certain real time control applications with iteration rates of 10 to 50 times per second. Other critical functions might take longer. The acceptance of recovery times of 1 minute or more essentially means that the program, which is to be recovered does not fall in the real time category.

7.2 Error Detection

The most fundamental conclusion that has been reached in the error detection area is detection of hardware failures

must be completely a hardware function. (We are confining our discussion to faults within the internal structure of the multiprocessor. Peripheral I/O devices can, depending upon their characteristics, employ central processor software to provide diagnostic capability.) The above conclusion is based upon the following reasoning:

a) An important aspect of any system which is to recover from a fault is to detect an error within a period of time which guarantees that the error hasn't propagated to a point where recovery becomes impossible. Assuming a given error is detected by a software self test routine, it is generally impossible to determine what information in memory has been incorrectly modified. Without the ability to isolate the damage, repair cannot be effected and recovery becomes unattainable.

Hardware error detection mechanisms such as parity, comparators and specialized logic provide a continuous monitoring upon the system. Software test routines can only be executed periodically in time.

Error detection logic, properly designed, will more nearly approach the goal of instantaneous error detection which prevents the propagation of failures.

b) If software self-test were to be employed one must consider the question of how long it will take to execute. Hardware error detection need impose little if any overhead upon the system performance. Software can spend a considerable amount of time for two reasons:

1) To be comprehensive an extremely large number of tests must be run.

2) They must be executed at a high frequency.

The unfortunate thing about software self-test in the past has been that, in most cases, hardware was not designed with self-test in mind. It was very difficult for the software to control precisely the hardware state. Micro level diagnostics tend to alleviate this problem to a degree. Because of an inability to test easily all features of a system, self-test software demonstrates the phenomenon that a large percentage of equipment functions can be tested with a relatively small amount of code, while the final few percent of the equipment tests require a very large amount of code.

c) The periodic nature of software error detection makes transient error detection difficult. Two categories of transients may be isolated:

1) Type 1 transients cause a temporary incorrect electrical signal but do not change the state of any storage element.

2) Type 2 transients occur at such a point in the sequencing of a processor that incorrect storage occurs. The hardware satisfies all tests that can be invented, yet bad information may exist which will eventually cause incorrect system performance.

If a type 1 transient is not detected it hardly matters to the functioning of the system. However, an undetected type 2 transient could possibly be catastrophic. An error detection philosophy which provides a continuous monitoring at critical points is necessary in order to prevent type 2 transients from going undetected and propagating.

Micro diagnostics, although more comprehensive and easier to write than software, must still be executed on a periodic basis. Their ability to detect transient failures must be seriously questioned.

d) The final point against software diagnostics as the sole error detection mechanism is that failures can occur which disable the execution of the software. Therefore, the signalling of fault condition can not occur.

7.2.1 Implementing Hardware Error Detection

Error detection is intimately involved with the specific failures modes of devices and equipment. If the various failure modes and the propagation dynamics of the failures are studied, then, in specific instances, the addition of a moderate amount of logic can detect the anticipated failures. On the other hand, one would like to employ techniques which are not very dependent upon the specifics of the equipment in order to provide a degree of flexibility and generality. The appropriate decision between specialized and generalized error detecting logic is a matter of engineering judgement.

7.2.1.1 Processing Unit: The processing units of the multiprocessor are the major sources of error propagation. If incorrect write operations are executed, due to a failed component, then the normal sequencing of the processing units, using this incorrect data, can cause propagation of the error to other portions of memory. Propagation of errors can extend beyond the multiprocessor system, if incorrect I/O commands are issued and executed. Because of the potential devastation caused by a processing unit failure, a maximum design effort must be undertaken to detect P failures before they propagate to other parts of the system. Within the limits of practicality, an effort must be made to detect almost all failures within P, before incorrect write operations or invalid I/O operations are executed.

Based upon these objectives, the study conclusions suggest that processing unit error detection be accomplished by employing two synchronized but independently operating processors with a fail-safe comparator placed across the memory interface. Some of the reasons for this conclusion are presented below:

a) Periodic software self-test cannot catch all failures before they propagate to multiple errors.

b) Error detecting codes internal to the processing unit cannot detect a large category of failures. For example, the failure of a control signal can cause almost every bit in a word to be incorrect. The use of arithmetic codes, such as a Modulo 3 check, produces inconsistent results under operations such as AND, OR, Not.

c) It will require at least twice the logic, and incur more than twice the cost, to detect all possible single component failures in P. Therefore, the cost of a dual P unit is reasonable.

d) The redundant processors can be packaged separately with independent power distribution. This will more closely meet the failure independence assumption.

e) Redundancy with a comparator at only one interface will reduce the number of interconnections between the redundant processors.

f) Errors are detected before bad outputs may propagate from the P. The comparator placed at the output of P might allow an error to propagate within P, but no bad information leaves P.

g) If one were to design a processor considering error detection as one of the main specifications, then each module could be designed to detect its own errors. Appropriate design efforts must be spent in maintaining statistical independence between failures and preventing errors in the error detection logic itself from going undetected. This innovation to the logic design effort would prove to be an interesting research topic. As far as employing the present SUMC design as the processing element of the multiprocessor, the use of two SUMC elements with a comparator seems to be the most reasonable approach.

7.2.1.2 Memory: The irregular structure of the processor leads one to consider the use of dual processors as a cost effective error detection mechanism. Memory structures tend to be very periodic in nature, possess little if any combinatorial logic outside of the addressing area, and therefore, are more amenable to the use of error detection codes. Simple word parity is a degenerate case of an error detection code.

Memory can be a significant contributor to the hardware cost of a multiprocessor system. For this reason, techniques other than brute force duplication of memory modules should be considered for error detection purposes. Depending upon the details of the construction of memories, different techniques can be employed. The following suggestions are made and seem to serve the purpose for most state of the art memory architectures.

a) Word parity can detect single memory cell failures, sense amplifier failures, and other failures which manifest themselves as single bit errors.

b) The incorporation of parity upon the address of the word proves satisfactory in detecting the failure of a single bit in the memory address register.

c) Employment of special current threshold circuitry can detect the simultaneous selection of more than one memory word at a time.

d) The use of a time-out indication can detect the failure of a memory module to sequence.

e) The use of a write-and-verify mode of operation, where every word written into memory is immediately read again, can verify correct storage. This is particularly applicable to NDRO type memory structure. For a DRO memory system one must face the problem that the

read operation which is used for verification must be followed by a write-for-restoration of the data. A failure can occur during the second write operation which would go undetected until the stored word is used again. However, the write-and-verify operation is still useful in detecting failure modes associated with transient addressing, control or bit storage failures.

f) Integrated circuit memories possess enough redundant addressing logic so that a partitioning of the memory into independent bit planes allows word parity to detect a large number of addressing errors. Present state of the art integrated circuit memories contain address decoding on each memory chip. Chips can be configured to contain one, two or four bits of 1024 words on each chip. Since each chip contains its own address decoding, a failure of a chip can only manifest itself as an error on the output of the chip itself. That is, it is localized to a few bits of the word. If each chip contained only one bit of each word, then a single word parity bit would detect all address decoding failures.

g) The use of separate read and write logic in the control area of the memory module will prevent a read command from turning into a write command, due to a single component failure.

7.3 Recovery

When a module of the multiprocessor fails, the presence of a spare (physically identical module) which can execute the same function does not necessarily mean that recovery can be accomplished. A failure not only eliminates certain physical resources (hardware) from potential allocation to executing processes, it also destroys information (program, data and status), which is required for execution. The major problem associated with recovery is not the necessity of providing spare hardware with an appropriate reconfiguration switching mechanism. It is, instead, the problem of re-establishing all the information required by the process to recover. In order to achieve recovery, the system must be returned to some past state which is known to be correct.

What exactly determines the state of a system? If real time is ignored, for the moment, then the system's state can be defined to be represented by the contents of all the storage elements, including M1, M2 and the Processor's control flip

flops. The more dynamic changes to a system's state are contained within M1 and P. M2 possesses less dynamic changes with time. M3 is even more static. As one proceeds from the more dynamic to more static elemtns of a system's state, time becomes less important to the recovery process. therefore, software, which is more time consuming than hardware, can be employed.

The discussion on recovery will address three major areas:

a) The processing unit, P and M1

b) Operating memory, M2

c) Input output controller (I/OC) and its channels

Suggested approaches to recovery from both transients and permanent failures in these three hardware areas are presented.

7.3.1 Processing Unit (P-M1)

7.3.1.1 Restartable Instructions: One of the main suggestions generated by this study, relative to a recovery from a processing unit failure, is to design all instructions to be restartable. This means that the point of recovery is the instruction during which the failure was detected. It is assumed that all failures are detected essentially instantaneously so that propagation of the failure does not cause incorrect information to be written into M2 or bad I/O commands to be executed.

Although a restartable instruction is not a difficult technical feat, it does require a design effort. The following ground rules must be applied during the design implementation of each instruction:

a) Each instruction must be partitioned into two phases. During phase 1 the instruction is fetched, data is read, computations are made and all memory write operations are placed into a temporary buffer area for execution during phase 2.

b) During phase 2 the buffered information is copied into its final destination in M1 and M2. The contents of the buffer area are not destroyed unti all the copy cycles are completed and verified. Each phase is designed to be separately restartable. Figure 7.1 schematically represents the execution of a generic restartable instruction.

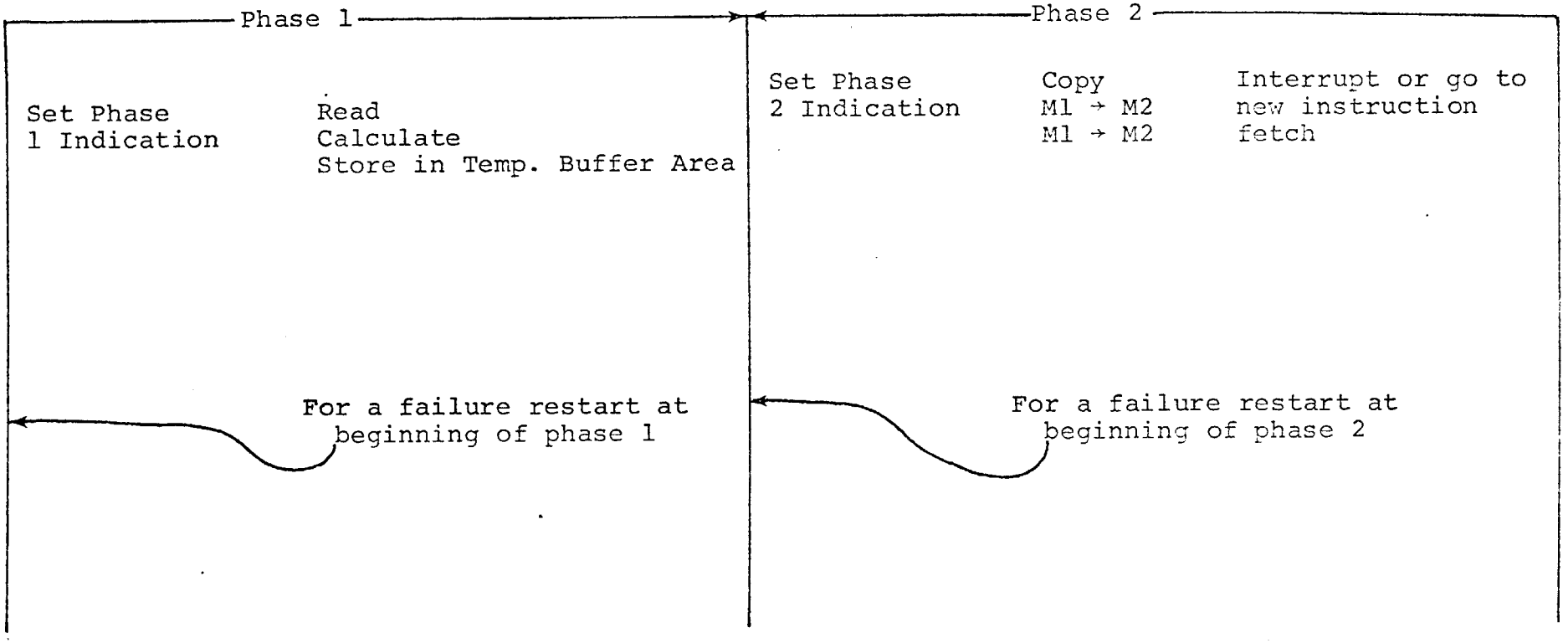


Figure 7.1: A Generic Restartable Instruction

c) If a failure indication occurs during phase 1, then the old copy of the program counter indicates which instruction was being executed. All of the information needed to execute the instruction has not changed, so phase 1 can be re-initiated. If a failure occurs during phase 2, then, even though some information might have been copied, the information temporarily buffered in M1 is still valid, and a complete re-initiation of phase 2 is indicated.

d) Interrupt testing can either occur at the end of phase 2 or at the beginning of phase 1. It is assumed that all interrupt conditions are caught in latches, so that the interrupt test is just a matter of reading these latches and determining whether to fetch the next instruction in the instruction stream or to enter the interrupt control micro-routine. The interrupt control micro-routine must be designed to be restartable and it must incorporate the concepts of a double phase operation with a buffer area, i.e., the interrupt control micro-routine can be considered to be a restartable instruction.

What does a restartable instruction design allow the system to do?

a) For transients which interrupt the normal execution sequence, but do not destroy data, the retry of an instruction will provide a simple method of recovery.

b) For transients, where information is modified, the information must be restored before the instruction is retried. The restoration of the lost P or M1 information can be accomplished by either error correction codes or by duplexed storage.

It is proposed that each instruction be designed so that after an instruction is executed, the state of the processing unit is always contained within M1. Each processing unit would contain two M1's so that in the event of failure of one, the information contained in the second could be used. The size of M1 should nto be more than 100 words and so its duplication presents little hardware impact.

c) Recovery at the instruction level allows the entire operation to be independent of the application programmer. Hardware and operating system primitives can determine when and how to restart. All considerations are based upon detailed information below the

instruction level. The application programmer could not care less about these details.

Because single instruction restart (SIR) allows a very quick recovery mechanism, one is not even concerned about the impact of the delay between error detection and recovery. This should be well within the iteration period of the highest frequency periodic application function.

d) Error detection within the instruction cycle as well as SIR tends to eliminate questions of error propagation and the interactions between a failure and the informational content of the rest of the system's storage.

7.3.1.2 Critique of Alternatives: Why the emphasis upon a restartable instruction? What are the alternatives?

a) In a batch processing system where multiprogramming is not used, the failure of a processing unit catches only one program in a running state. All the submitted programs are completely independent and recovery is simply a matter of reloading the program and data. Many functions on the space station can be handled by this "fresh start" approach. It is simple and imposes minimum overhead.

However, the real time aspect of some of the space station processing requirements makes the "fresh start" approach unfeasible.

b) A "checkpoint restart" approach to recovery has been applied to systems where problems requiring hours of computer time are being run. At fixed intervals the complete contents of core as well as the processor registers are dumped onto a back up area on disk or tape. A snapshot is taken of the system's state.

A superficial look indicates that with a 1 μsec cycle time and a 100K word memory, a memory dump can be accomplished in 100 milliseconds. This is not an unreasonable time. However, let us investigate the implications of "checkpoint restart" a little more deeply.

1) If a snapshot requires 100 ms then one must consider its effect upon system throughput. If one desires to limit the overhead imposed by this

function to less than 5%, then a snapshot can not be taken more than once every 2 seconds. If real time requirements allow a recovery time of 2 seconds, then "checkpoint restart" might be a viable candidate.

2) If the contents of operating memory and the processing units are rolled back 2 seconds in time, can one guarantee that the state of the mass memory is always consistent? Must the contents of mass memory also be dumped when operating memory is dumped? In general, the answer is yes. In a virtual memory system where memory hierarchy must not contain inconsistent information. Dumping M3 periodically onto some archival storage device such as tape (M4) seems to eliminate check point restart as a valid candidate for recovery in a real time environment.

3) Even though M2 can be dumped in 100 ms; a disk, drum or tape probably couldn't absorb the data at a rate higher than 10 MBPS. This will increase the snapshot time for 32 bit words to 320 milliseconds and the snapshot period to once every 6.4 seconds.

7.3.2 Recovery From an Operating Memory (M2) Failure

Hardware failures and electrical transients in memory systems cause information to be destroyed. Recovery from a memory failure would be very easy if the error patterns caused by failures and transients could be known with certainty. Many error patterns could then be corrected by employing error correcting codes. Unfortunately, it is impossible to analyze all possible failure modes under all possible environments to determine all possible error patterns. Failures exist which can not be corrected by error correcting codes. Error correcting codes are not useful when the timing mechanism fails in such a way as to prevent memory access. A failure in the addressing mechanism can not be corrected by the encoding of data.

Error correcting codes can be successful when the predominant error modes are single bit failure or small burst failures. In general, however, duplication of the information contained within the memory cells is required for successful recovery from an M2 failure.

7.3.2.1 Problem Areas: When attempting to design a system which is recoverable from M2 failures, a number of distinct problem areas must be resolved:

a) Memory Management

Normal (non-failure-tolerant) memory management deals with the allocation, deletion, and control of memory space for program and data entities. When failure recovery is made a requirement, additional questions arise; how to deal with redundant storage of critical information? How shall the hardware and software interact to:

1) enable the continuous storage of redundant information?

2) allow the accessing of valid information in the presence of a fault?

b) Hardware Fault Isolation

When a memory error is discovered, how can it be isolated to a repairable piece of equipment?

c) Information Fault Isolation

If the failure is isolated to a specific memory module, one must be able to determine what information was destroyed so that recovery action can be controlled.

d) Storage of Redundant Information

Since the redundant storage of information becomes a necessity for critical programs and data, a question arises as to how and where the redundant information should be stored; in M2 or M3 or a combination of both?

7.3.2.2 Factors Behind the M2 Recovery Approach: A number of considerations pointed to the suggested M2 configuration. The following items consist of assumptions, observations and the philosophy which leads to the approach presented in the next section.

a) Consistent with the processing unit's failure recovery philosophy, the applications programmer should not be concerned with the details of the recovery procedure. This is handled by the hardware and operating system. There is however, one aspect that must involve the application programmer. He is the only one who can initiate the specification as to which program and/or data segments are critical. By definition, critical

segments are all those segments used by programs which must recover and continue execution after a failure.

Non-critical programs need not recover. They must, however, be terminated in such a way so as not to interfere with critical programs. This is called Fail Safe. Some observations and requirements necessary to enable a program to Fail Safe are presented in Section 7.4

Once the applications programmer indicates the programs which are critical the compiler can statically assign critical or non-critical status to segments it creates. Similarly, the operating system must also assign criticality status to segments it dynamically creates: For example, a stack.

b) The recommended approach to memory management is to employ a segmented virtual memory system.

The virtual memory approach allows an exploitation of the difference between read-only (program and fixed data segments), and read-write (variable data segments) information. If an M2 module which contains program segments fails, it is desirable to exploit the virtual memory mechanism, already implemented within the system, to aid in the recovery process.

Most program segments can be considered to reside in M3. They are brought into M2, on demand, for execution. If the program segments contained within the failed M2 module were, as the result of the failure, made "not present", then the M3 to M2 transfer mechanisms will allocate space and transfer anew the required segments automatically. The "not present" segment indication is contained within the program segment descriptor. Descriptors are considered to be data and are in turn stored redundantly in M2.

c) For a large computational system on-board a space station, it is reasonable to assume that repair or replacement of a failed M2 module will be performed relatively quickly. The hardware error detection mechanisms should be able to isolate to a repairable unit, and to indicate the action to be initiated by the software.

However, there must be sufficient M2 space available so the system can run without "thrashing". This entails modifying the work load so as to reduce the memory required to accomodate the working sets of the

remaining processes. Possibly, the number of processes of particular types might be limited to reduce the work load.

7.3.2.3 Proposed Configuration for M2 Failure Recovery: The proposed configuration defines an M2 module as four M2 units which are interleaved on their low order address bits (see Figure 7.2).

Information segments may either be stored in a simplex or duplex mode. The mode is specified within the descriptor. Most program code would be stored simplexed and interleaved across the four memory units. Most critical data segments would be stored duplexed. In the duplexed storage mode address i and i + 1 contain identical information. That is, two adjacent memory units contain identical copies of the redundant words.

A minimum of two memory ports connect to the redundant P interfaces. Communication with any M2 unit can occur through either port. This is under control of the command issued from the processing units.

M3 is used to backup most program segments. M2 is used as the backup for data and certain critical program segments. Program and Data Segments can be stored anywhere in M2. When space is assigned to a critical data segment, a double size "hole" must be found in M2. This does not impose any extra effort upon the memory management function.

Redundant writes into independent units of M2 are accomplished automatically via the dual redundant processing unit bus links. Recovery of M3-backed-up information requires making the segment "not present". The memory management routine which handles segment faults will automatically reload the M3 segments when required, on demand.

Whenever an M2 error is detected, the error indications are communicated to both halves of the processing unit so they can continue to perform identical operations. The ability to restart an instruction can be exploited in attaining system recovery after an M2 failure. As soon as an M2 error is detected, the processing unit traps to a special micro-routine which bootstraps into the sequence indicated in Figure 7.3. After recovery, the instruction which was terminated by the trap can be re-executed (if the M2 error was detected during ϕ_1 of the instruction) or the instruction may be completed (if the M2 error was detected during ϕ_2 of the instruction). It is interesting to note that M2 read operations occur only during ϕ_1 while M2 write operations occur only during ϕ_2.

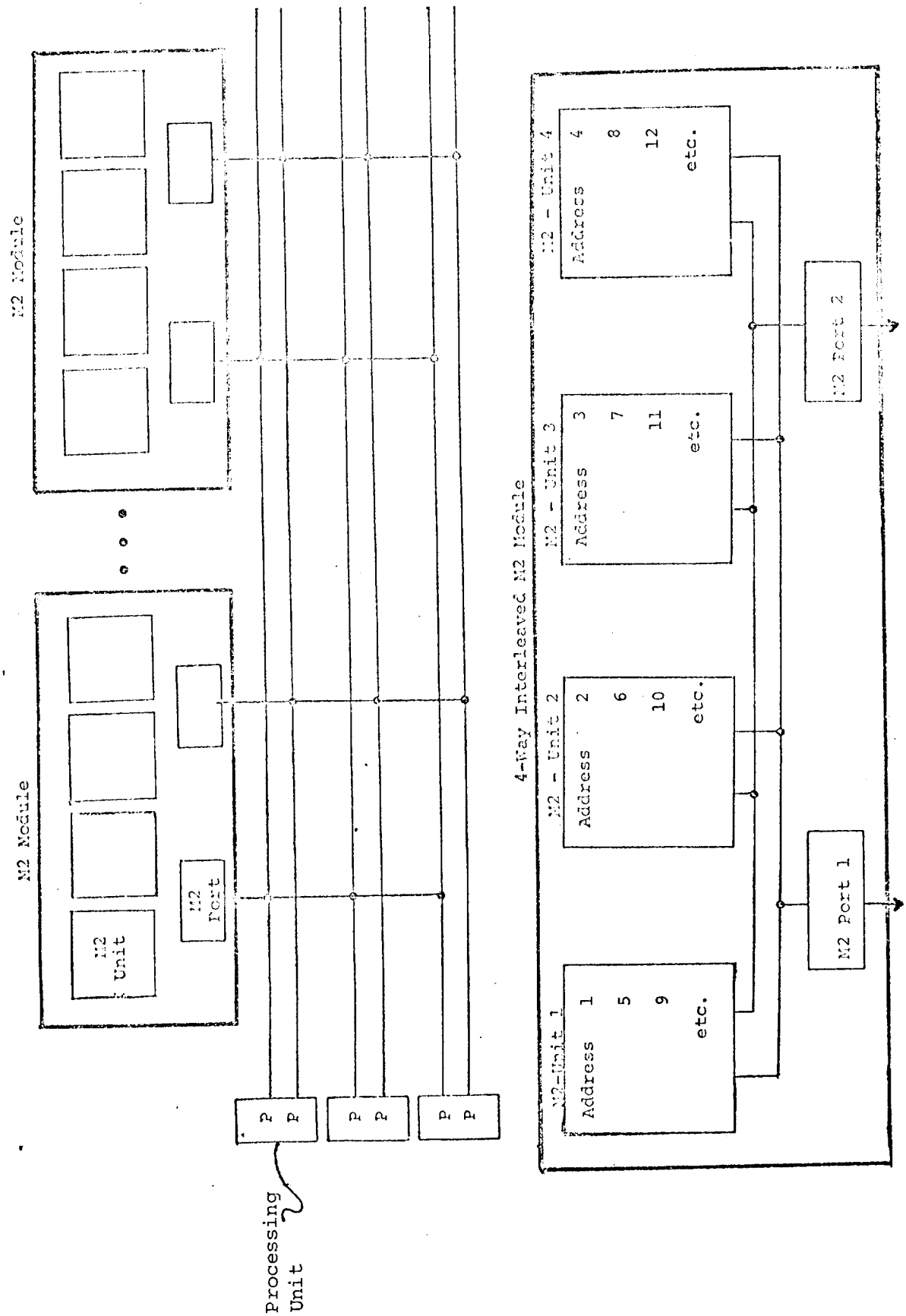
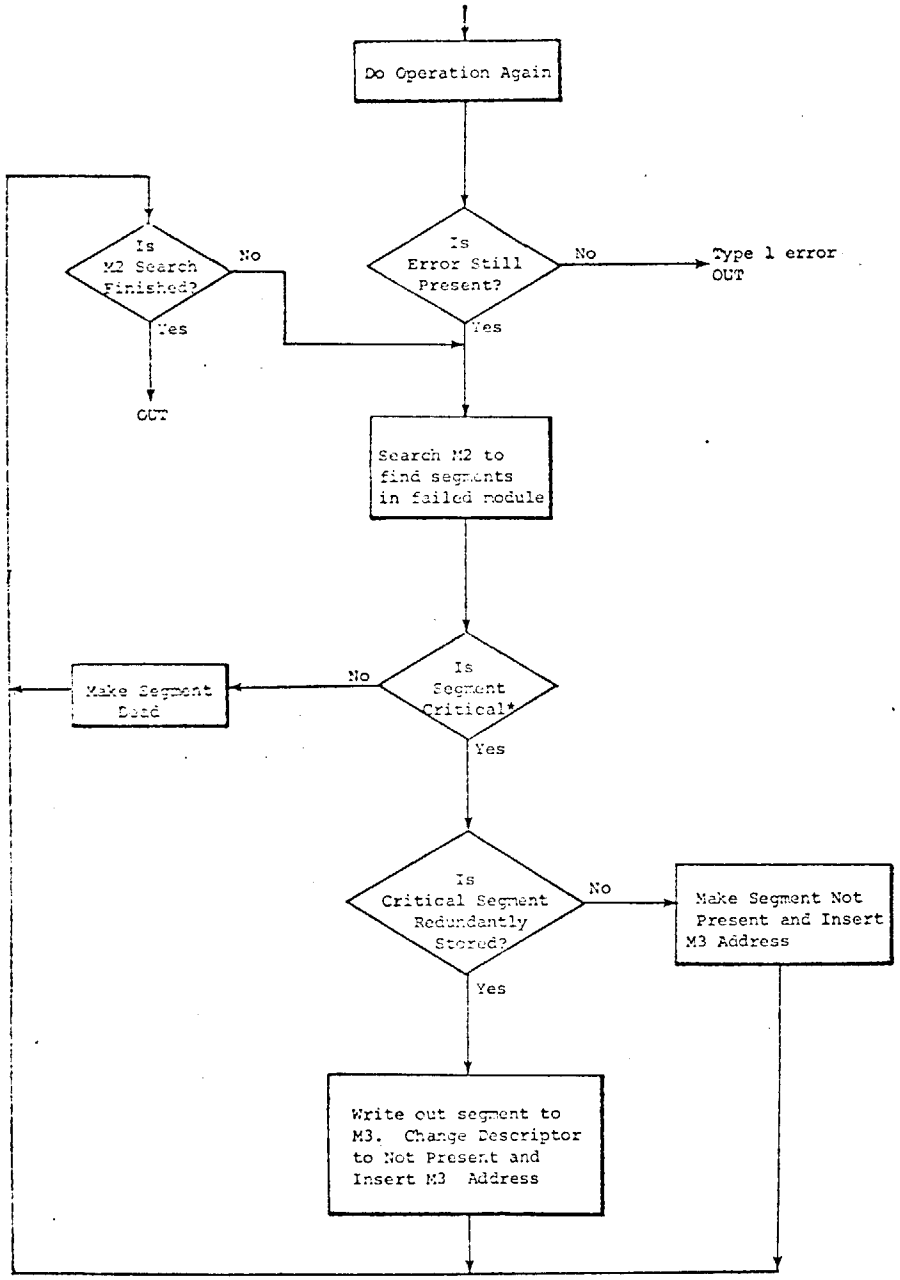
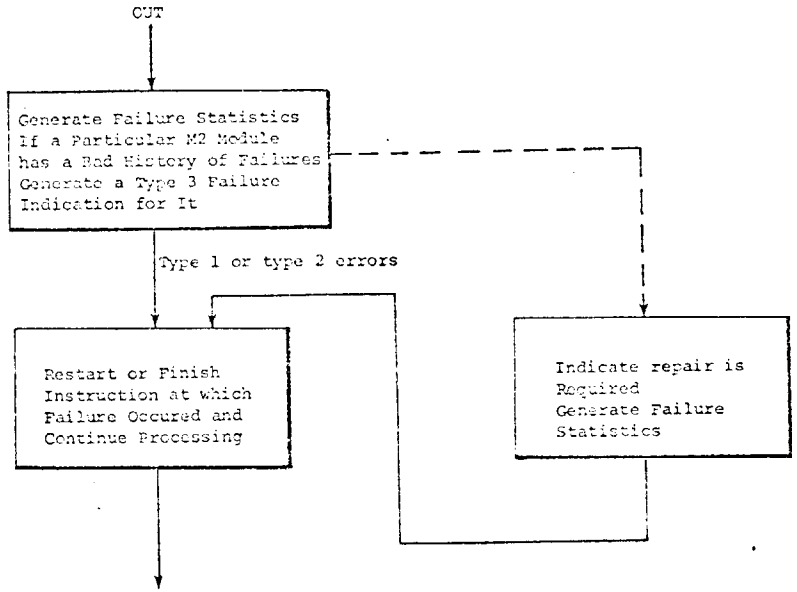


Figure 7.2: Interleaved Memory Units





Legend:

Type 1 errors - Transients which do not change the contents of memory

Type 2 errors - Transients which do change the contenst of memory

Type 3 errors - Permanent hardware failure which necessitates the repair or replacement of a memory module

* A Critical Segment is associated with a restartable entity. It possesses a backup on either M2 or M3

Figure 7.3: M2 Error Indication

When an M2 error indication is first recorded, the M2 operation will be tried again. If the error does not recur then a type 1 error is indicated. However, if the error indication persists a search is made to determine which segments are stored in the suspect unit. To accomplish this search in the presence of a failed unit, the header word containing a pointer to the segment descriptor as well as a link to the next segment is redundantly stored. Figure 7.4 shows the storage allocation for both simplex and redundantly stored segments.

All non-critical segments within the suspect module are put into a "dead" state.

Critical segments can be either redundantly stored or not. A redundantly stored critical segment is written out to M3 so normal memory management can be used to allocate new space for it when required. Since it is assumed that failures do not simultaneously affect both copies of redundantly stored information, the good copy can be accessed after a failure.

Non redundantly stored critical segments are made not present. Fixed data and programs fall into this category.

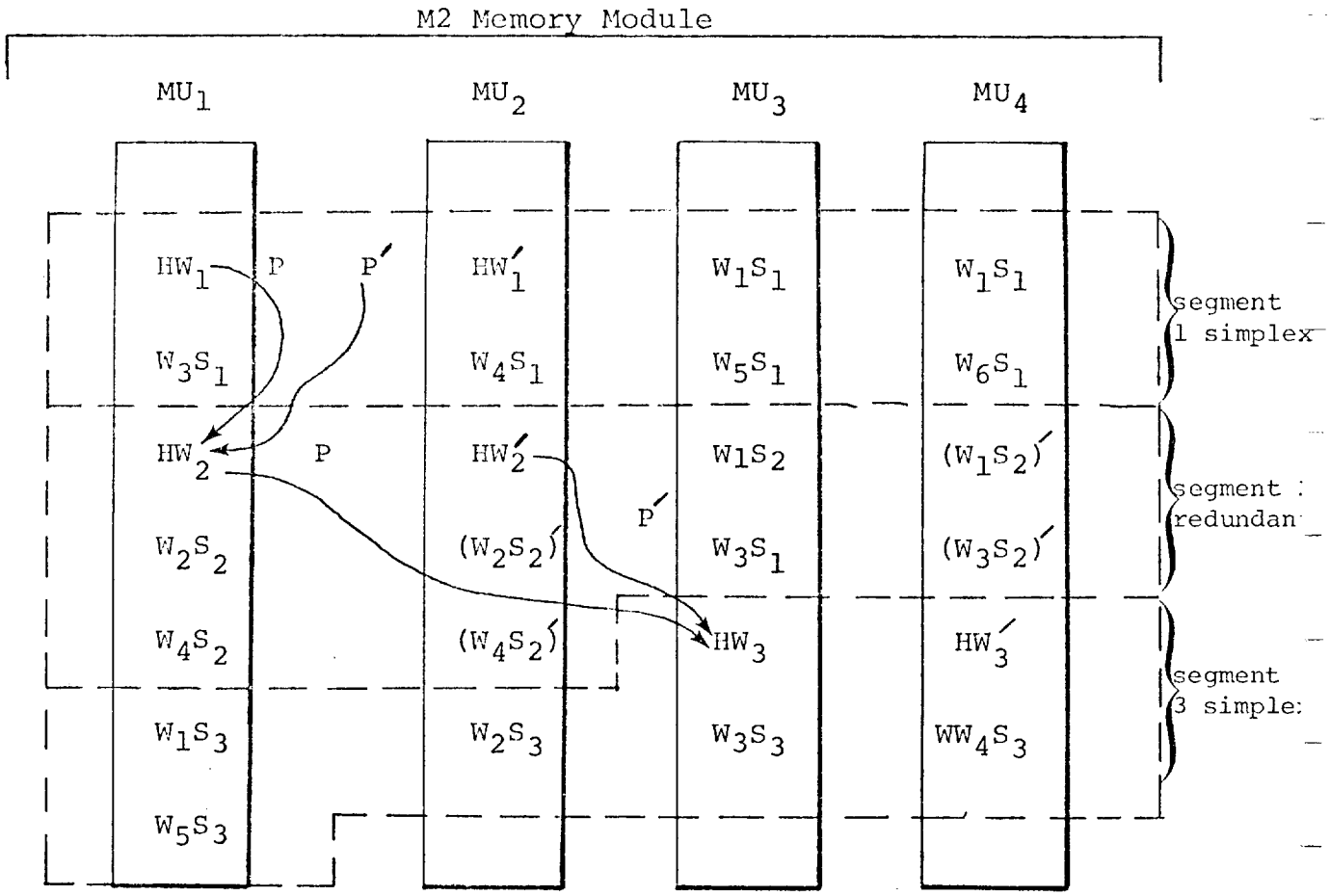
For all M2 failures, statistics are maintained indicating a failure history. If an M2 module develops a bad history of failure, then it will be removed fron an active status. The definition of how many failures within a given time period indicates a bad history, can be considered a design parameter depending upon whether transient or permanent hardware failures constitute the predominant failure mode.

7.3.3 Fault Tolerant Aspects of the I/OC, Channel

This section will address problems associated with recovery from a transient or permanent failure in the I/OC or communication channel between the I/OC and the device.

Many constraints must be placed upon the I/OC, channel and the attached devices and controller. Figure 7.5 presents schematically the elements which will enter into the discussion. Only one I/OC, channel and device is shown. Clearly more exist in a real system. Our discussion will focus on only one I/OC, channel and device at a time.

7.3.3.1 Incorrect I/O Commands: The basic recommended approach is to eliminate the possibility of executing incorrect I/O commands. As a general principle all I/O devices require some degree of feed back to the MP, if any fault tolerant design goals



Legend:

MU_k = k^{th} memory unit

HW_i = Header word of i^{th} segment

W_jS_i = j^{th} word of the i^{th} segment

HW_i' = redundant copy of HW_i

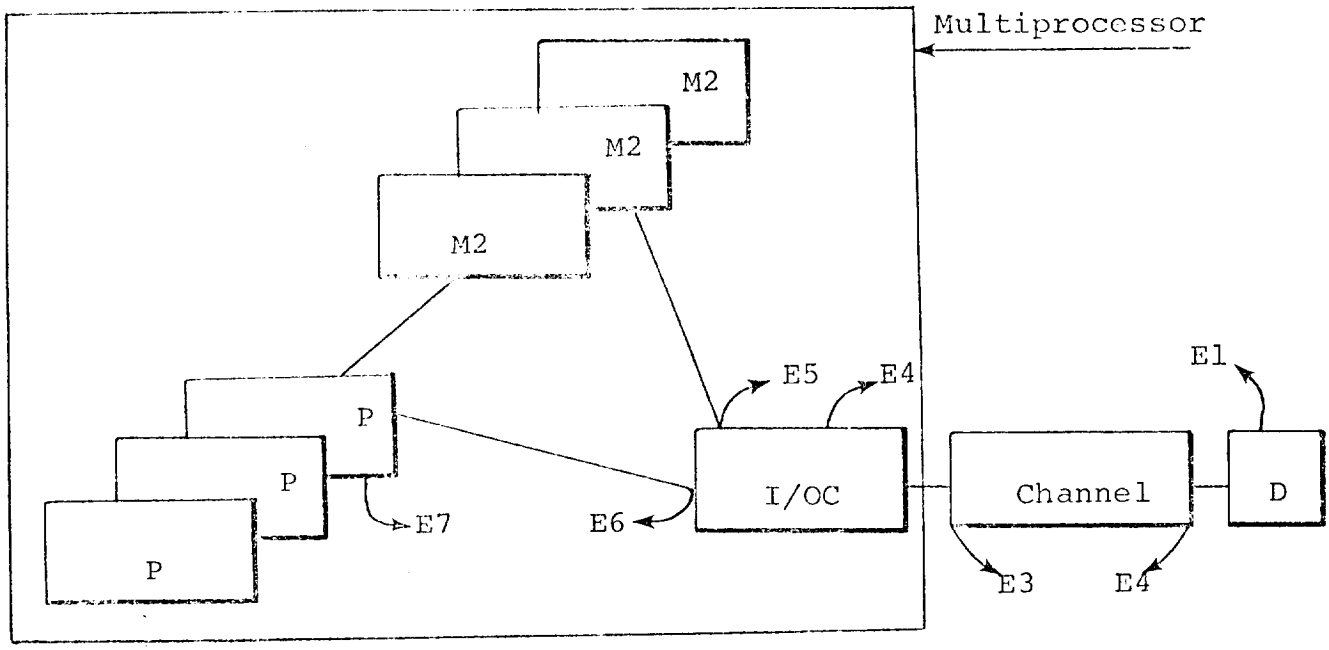
$(W_jS_i)'$ = redundant copy of W_jS_i

P = pointer to start of next segment contained in HW_i

P' = redundant pointer contained in HW_i'

Figure 7.4: Storage Allocation in Interleaved M2

Figure 7.5: I/O Elements



P	Processing Unit
M2	Operating memory
I/OC	Input output controller
Channel	Communication Channel between I/OC and Device
D	Device and associated controller (if required), e.g., Printers, CRTs, IMU's, other computers, etc.
E1	Device error
E2	Channel error detected by device
E3	Channel error detected by I/OC
E4	I/OC error
E5	M2 error detected by I/OC
E6	Interprocessor communications error detected by I/OC
E7	Processing Unit error

are to be achieved. No external device can be allowed to run in an open loop mode without communications back to the I/OC.

One of the most devastating aspects of I/O failure is the possible execution of illegal unwanted I/O commands. A major design effort, which will impose constraints on the elements of figure 7.5, must be undertaken to eliminate or minimize the possibility of incorrect I/O. Let us look at a typical I/O sequence, with safeguards to minimize this possibility.

a) The processing unit issues an I/O command to the I/OC.

b) The I/OC reads the indicated M2 location to obtain the I/O descriptor.

c) The I/OC sets up the channel and issues the command to the device.

d) The device echoes the command back to the I/OC for verification.

e) If correct, the I/OC issues an execute sequence to the device. The device then executes the command which may require reading or writing into M2.

f) After execution, a finished indication is sent from the device to the I/OC and this status is set into the I/O descriptor in M2, or an interrupt is generated.

Let us investigate the effect of a failure during any of the sequential steps listed above. Error indications can occur from many sources including P, M2, I/OC, channel and device.

A failure indication, E5, E6, or E7 during steps a and b allows time so the I/O can prevent the issuance of the command. If an error, E1, E2, or E3 is detected during step c, then the I/O must also terminate the command, since an execute has not been issued to the device. An E failure indication during step c should result in an emergency sequence to cancel the I/O request already issued to the device.

The echo check, step d, provides a positive verification (feedback) that the device has successfully received the command.

An I/OC or channel failure indication during the execution of a command must result in a sequence of operations which is very device dependent. This will be discussed in section 7.3.3.7.

7.3.3.2 Super Critical Commands: Although the I/O portion of the space station is inadequately defined, it seems reasonable to postulate the necessity for a small number of super critical commands with the following properties.

a) It is most disastrous if the command is executed when it shouldn't be.

b) It is better to abort the command or action if anything seems to be going wrong rather than execute it incorrectly.

Examples of such commands might be "Stage the Rocket", "Purge the Airlock", etc. What should be done if failure occurs during the execution of a super critical command? The answer is to make the command fail safe, by issuing it or a facsimile thru multiple channels to the device. Only when all the arming conditions for the command are properly set is the device allowed to execute. If any discrepancy is noted at the device, command execution must be held up for resolution by the MP.

7.3.3.3 Interrupts: In many instances, the system is faced with the problem of "phantom" interrupts or missing interrupts. Fault conditions within the interrupt logic can cause undesired interrupts (phantom interrupts) or can possibly prevent the generation of interrupts which should occur. The action to be taken by the system in these cases is very dependent upon the interrupt condition one is considering.

Let us consider two cases:

a) The Expected Interrupt

Often interrupts are expected when an I/O device command finishes. The exact time of occurence of the completion of the I/O command is not known, but the worst case time may be estimated.

A time out error indication is a simple mechanism which will inform the system that the I/O device has not finished executing the command or at least the "done" interrupt has not been received within a given time period. If the I/OC and channel have a sufficient amount of internal error detection, the failure can probably be attributed to the device itself.

The action to take might involve a limited number of retries of the operation or a call for system reconfiguration which eliminates the device from use.

If a "phantom" interrupt occurs, which indicates a device end condition for a device which wasn't being used, then clearly this interrupt should be ignored by the system. This feature can be incorporated into the interrupt handling routines.

b) Unexpected Interrupts

These are a class of interrupt conditions which are provided for but which are unexpected. For example, the failure of a P or M2 unit might cause a different P to get interrupted. If this failure interrupt is signalled when the condition really doesn't exist, it is probably still wise to service the interrupt rather than ignore it. It is better to configure into a degraded mode of operation, for a short while, when it isn't necessary, rather than not to reconfigure when it is necessary.

Other interrupts which are unexpected are not associated with failures. Many are traps, such as absent segment trap conditions. The servicing of an absent segment trap condition when one doesn't exist can lead to inconsistent situations and ultimately system failure.

One design feature, which can be applied to certain I/O interrupts, involves a handshaking or interrupt verification concept. This feature would have the system verify that the interrupt which was signalled really does exist. The device which signalled the interrupt must retain the interrupt condition information until after the verification cycle. The verification can either be performed directly by the I/O unit or by a processor through an I/O command.

7.3.3.4 Non-State-Dependent Sequences: If an I/OC or channel sustains a transient, which causes the termination of an I/O sequence, then it would be desirable to rely upon a recovery policy which would cause the reissuance of the I/O command. In order for this recovery policy to be satisfactory, the response of the I/O device to the command must be only a function of the command and not of the state of the device itself. This feature can be designed into the device if one is careful about the initial design specification and the type of commands one allows.

For example: Assume a tape unit is at the end of record 6 of file 1. A command which says "Read the next record" is very dependent upon the state of the tape unit; namely the position of the tape. A better command structure would be "read record 7 of file 1". The result of this command will always be the same independent of the position of the tape.

It should be clear the "Read the next record" would not prove to be a satisfactory command to reissue in case of a failure in the middle of reading record 7. Record 8 could be accessed instead of 7.

7.3.3.5 Complete Message Buffer: If errors can be detected as soon as they occur and if recovery from transient errors is required, both the I/OC and the device must have enough buffer storage so that a retransmission of the entire message (data and command) can be made. The I/OC buffer may, indeed, be M2 and the buffer storage element of the archival memory might be the tape itself.

It is undesirable to have to recreate the entire message because of a channel transient error. Retransmission appears to be a reasonable approach.

7.3.3.6 Real Time Aspects: When the MP is used as an element of a real time control loop, outputs can be required periodically. If a failure occurs during a real time I/O command, the device could possibly have to wait for a number of iteration cycles for the recovery cycle to be complete.

In this instance, the device must be provided with a capability to extrapolate from old updates until the system has recovered. This might require nothing more than assuming the last update is still valid. Possibly, more complex methods are required.

7.3.3.7 Failure During the Execution of an I/O Command: If a transient occurs, the actions to pursue in order to recover become extremely device dependent.

Consider the following examples:

a) Many of the external devices attached to the space station data bus are transducers, to monitor temperature, pressure, gas mixture, etc. If an I/OC or channel failure occurs, the appropriate action for recovery would be to ignore the results of the command in progress, clear the buffer or reset the device if necessary, and reissue the command.

Any non-destructive read operation can be reissued for recovery purposes. Destructive read operations should be eliminated from the system specification or temporary redundant storage or redundant devices must be employed.

b) Consider the case of updating the refresh memory of a CRT output device. Assume a failure occurs during the update operation and the possibility of incorrect information on the CRT exists. Recovery action can consist of nothing more than reissuing the update command. If recovery takes 100 ms the human operator might only notice a small flicker on the screen and no damage is done to the overall system.

c) Consider the case of a printer. Assume a failure occurs in the middle of a print cycle. It should be clear that the reissuance of the PRINT command is inappropriate for recovery since the old printed output, possibly incorrect, would exist immediately on top of the new valid printed output. Page boundaries would be incorrect. Before reissuance of the print command, the page must be spaced. If a plotter instead of a printer were being used, the computer operator would have to be informed to insert a new sheet of paper in the plotter.

d) Inter-Computer Communication. Quite possibly, the space station will contain pre-processors in addition to a large central multiprocessor. Pre-processors are employed so as to buffer the high bit rate of the device. (See Figure 7.6.) They perform high frequency interactive calculations and provide a data rate reduction for the system.

Unlike simple input output devices which can recover with reissuance of commands, a pre-processor reaction to a command can be very dependent upon its own state.

All the concepts of command verification and message buffering, must be built into the pre-processor. The programs in the pre-processor must also be designed to run asynchronously from the multiprocessor.

7.3.3.8 I/O Locks: When a software process requires access to an I/O device, the device may required to be locked to the process. That is, no other process can access the selected device until the previous I/O request is finished. Problems of deadlock exist when the initiating process fails.

If the software process recovers quickly enough, then the lock does not remain on the I/O device for an excessive time. However, if recovery takes a long time or if the process is specified to be non-critical (that is it need not recover), then some mechanism must be designed into the system to release the I/O lock. This is one of the elements to consider in allowing a process to fail safe.

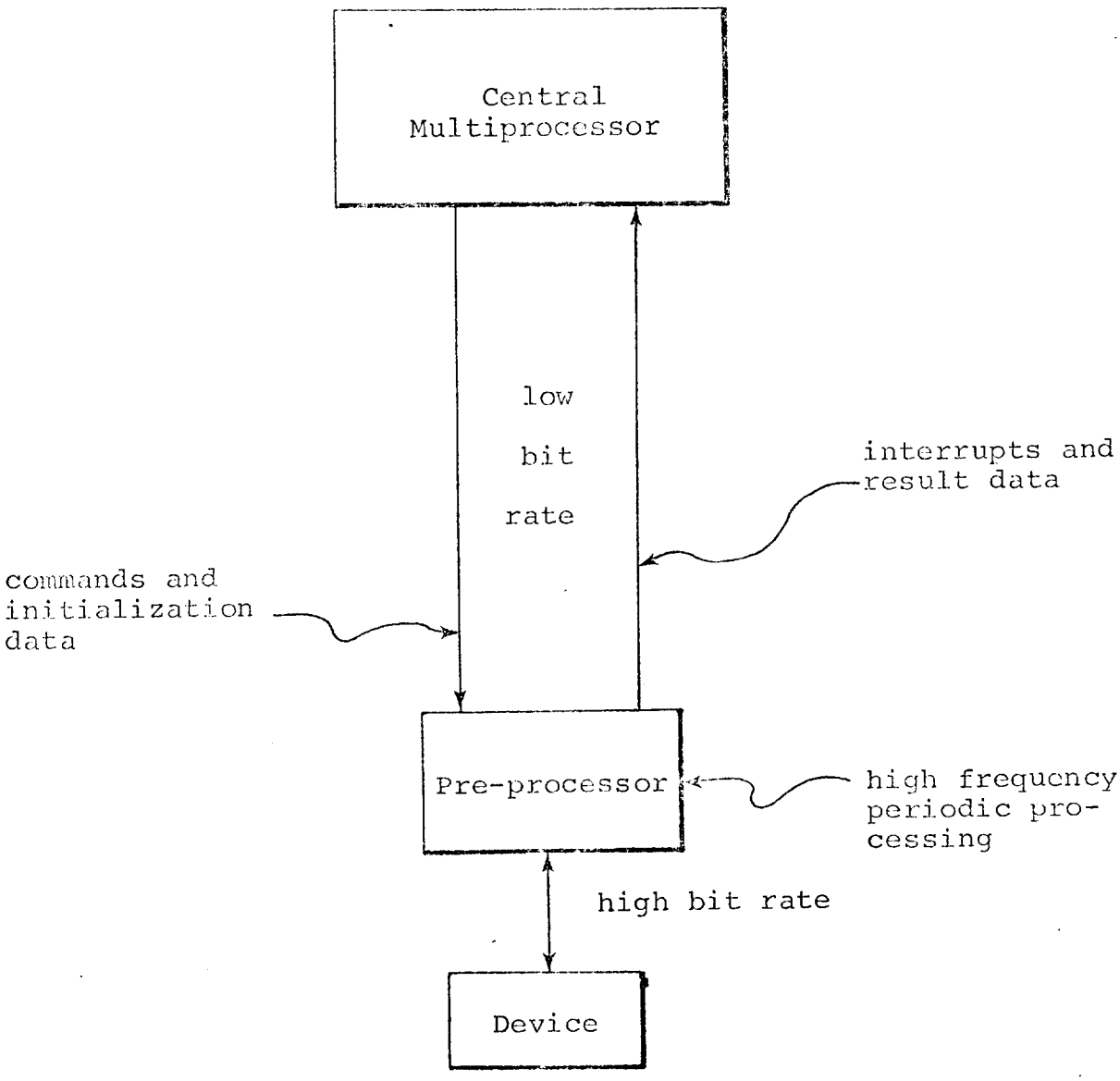


Figure 7.6: Pre-Processors

Even though the process need not continue operation, in case of a failure, a special I/O routine must be executed to search, find and release all locks created by the process which terminated.

7.4 The Implications of Fail Safe

Although it is the physical hardware that fails, it is conceptually useful to consider the process being executed at the time of failure to have failed. Only one process can fail when a processor fails. In the case of an M2 module many processes can be affected.

It is assumed that in the space station environment all processes are either required to recover or fail safe. None are allowed to be abruptly terminated without consideration of the interaction between the termination and the rest of the system.

A number of problem areas arise when one considers the implications of Fail Safe. Some of these are discussed below:

a) In order to maintain system throughput in a multiprocessor, the intrinsic parallelism within a function must be exploited. Parallel processes are spawned and executed simultaneously on different processors.

   If a process is to fail "safe", all the fork points which were created must be examined and all the spawned processes terminated. This feature must exist within the executive function of the system which controls the termination of processes.

b) If a process is to fail "safe", all the I/O commands issued by the process must either be cancelled, terminated or completed. None may be left indefinitely on queue. The various commands issued to each device must be studied to ascertain the effect of a premature termination of the issuing process. If a tape was in the middle of reading a record, the read cycle can be completed. Upon receipt of the "Done" indication, the read data can be discarded. If a command is still on an I/O queue, it can be cancelled. If a device is being written into, it is not clear that the write operation can continue when the initiating process is terminated. All these types of questions must be considered for each I/O device when one desires a process to fail safe.

c) When a memory unit fails and a segment of a non-critical process is made dead, questions must be raised as to the

disposition of the other valid segments within M2 and M3 associated with the failed process. The following suggestion is made:

One of the conditions which will cause a process to be killed, will be when it attempts to access data contained within a dead segment. When this occurs, control will be transferred to an executive routine which will control the operation of systematically terminating the process. This includes:

1) Placing the process in the dead state

2) Placing all spawned processes in the dead state

3) Releasing the stack number (in the case of a stack machine) and the space used by the process and dependent processes.

Contained within the process stack are descriptors of all the local data segments currently being used by the process. The space used by these segments must eventually be reclaimed for other uses.

d) During the normal execution of the memory management function, any segment not referred to within a period of time will be replaced by more active segments. This includes any dead data segments that may exist. Eventually, all the dead segments in M2 will be overwritten by just letting the system run normally. However, it is possible for dead data segments to occupy space on M3 which could possibly be used for other segments or for file storage.

At some point a "Garbage Collection" routine will have to be executed in order to reclaim this lost space. Most probably, the normal reclaimation of fragmented M3, due to M3-M4 control, will provide the required service.

e) In general, the executive design must consider the actions to take when a process enters the dead state. If an interrupt is directed to a process which is in the dead state, it should be ignored and any other process which is dependent upon the dead process must be informed so that appropriate action can take place.

Chapter 8

CONCEPT VERIFICATION

8.1 Background

The multiprocessor (MP) system proposed for future manned space stations will employ many new concepts which will hopefully enhance the performance and reliability of the system. This chapter will discuss the validation of various concepts proposed for the space station MP. The concepts to which reference is made are not applications software or SUMC hardware but rather those aspects of the system which interact with applications software and SUMC hardware to control the operation of space station subsystems and experiments. One wishes to verify that the ideas which will be implemented do indeed yield the required performance with an efficient utilization of resources.

How does one go about validating a new concept, or at least establishing confidence that a given approach will prove satisfactory? The ultimate answer is to build the system, run it, and evaluate its performance. This of course is an expensive process, especially if many new ideas have to be frozen into a design before it is evaluated. In order to provide a more orderly, cost effective approach a two level simulation is proposed, both levels being carried out before the system is committed to operational use.

This chapter will discuss both a high-level and a more detailed low-level concept verification process.

a) The first verification phase involves both analytical techniques as well as a high-level computer simulation employing idealized work loads and environments. The results of this effort will verify that a given design concept can achieve specified qualitative goals.

b) The second phase involves a more detailed, low-level simulation requiring both simulated and actual hardware and software modules. The objective of this phase is to verify quantitative goals, by means of measurements and design modification.

As part of both verification steps, measurements are made and design parameters are modified so as to optimize system performance. The specific activities involved in the design verification and performance optimization of the space station multiprocessor concept will be presented in the remainder of this section.

8.2 Phase 1 -- Initial Analysis and High-Level Simulation

8.2.1 Objectives

The initial analysis and high-level simulation attempt to achieve the following objectives:

8.2.1.1 Design Features: The major design features must be established. In a MP system this will include:

a) A definition of memory management philosophy

b) The appropriate utilization of local memory

c) Interrupt and I/O analysis

d) The structure of the MP internal bus

For example, the application of simple analytical techniques will demonstrate the inappropriateness, from a performance standpoint, of a single 32 MBPS internal bus which is time-shared between P's and M2 elements.

8.2.1.2 Parameters: The parameters which should be made variable in the low-level simulation must be identified and segregated, so that performance can be optimized. For example, the simple analysis of local memory and its effect upon performance indicates that the major parameters are the M2/M1 speed ratio, r, and the hit ratio, h.

The isolation of these parameters is significant in that performance improvement or degradation is very sensitive to h. Clearly those hardware and software elements which control h should be made as variable and flexible as possible.

If a virtual memory is employed a simple, high-level simulation or analysis will show that the following parameters should be made variable:

a) The page size (if paging is employed).

b) The replacement algorithm.

c) If an associative memory is proposed, its size should be variable. Performance is very sensitive to the search time of the page location algorithm.

d) Possibly, the utilization of a variety of different access times to M3 devices should be considered.

e) The size of M2 could be a parameter. The "thrashing" threshold has to be established if software expandability is to be achieved.

The main objective of this effort is to isolate as many parameters of design as possible through a careful scrutiny of all major design features.

8.2.1.3 Assumptions; Another objective of this first phase effort is to establish clearly all the assumptions, implicit or explicit, that formed the basis of major design decisions. For example, why was a multiprocessor chosen? Three answers are possible:

a) A cost effective performance increase.

b) Reliability improvement through the use of identical elements and an ability to recover.

c) Expandability.

All three of these assumptions or desires drives one to the conclusion that the executive system, which interfaces the hardware and applications software must be generalized enough for expandability, yet it must be implemented in such a way as not to produce an excessive overhead. Reliability implies a comprehensive error detection scheme. Recovery implies a specific communication interface between the hardware and executive.

8.2.2 Tools for High-Level Simulation

8.2.2.1 Simulation in General: How does one approach the problem of developing a high level simulation? What tools are available? Reference 1 discusses techniques available for both macro (high level) simulation and micro (detailed low level simulation). Macro level simulation is concerned with abstractions

of computer systems which are designed to expose and analyze critical design parameters. Generally speaking, these simulation techniques deliberately suppress design detail, and concentrate on broadly defined measures of system effectiveness.

Computer simulation at this level has its basis in queueing theory, the probabalistic analysis of the interaction between users and facilities. The role of simulation is to exercise user and facility interactions whose complexity exceeds the bounds of known or feasible analytic solutions, by Monte Carlo methods.

Digital computer facilities have long exhibited the symptoms dear to the queueing analyst: namely, bottlenecks. The reader will probably have personal familiarity with situations where a data processing facility has become hopelessly inefficient due to one, or a combination of, bottleneck elements.

The objective of high-level simulation is to obtain an advance estimate of the performance of a computing facility at the design stage. To be successful, the simulation must anticipate the way the system would work if it were built. The successful simulation designer must accomplish all of the following steps:

a) He must satisfy himself that simulation is an appropriate analytical method, and that the elements of the system and the job stream are sufficiently defined.

b) He must verify that the results of the simulation are correct, and that they are appropriate to his purpose.

c) He must explain and substantiate his results and proselytize his conclusions in order to affect future events in a constructive way.

These generalizations are noted here because there seems to be an uneasiness among professional personnel about high-level simulation of computers. This is probably because the technique of simulation has been often misused, particularly by neglecting the fundamentals listed above.

8.2.2.2. GPSS: A generalized macro simulation language GPSS was developed by Gordon [2] of IBM. GPSS deals in transactions, events, facilities, storages, and queues. A transaction is generated for each element in the job stream. Events mark the movement of the transaction through the system of facilities, storages and queues. A facility is a system element that can accomodate only one transaction at a time. A storage is a

system element that can accomodate many transactions up to a specified limit, at a time. A queue is a waiting line. Gordon gives examples of these concepts as they might occur in different systems:

Type of System	Transaction	Facility	Storage
Communications	Message	Switch	Trunk Lines
Transportation	Car	Toll Booth	Road
Data Processing	Record	Key Punch	Memory

There have been at least two efforts to develop specialized simulation language for computer systems. These languages are CSS II [3] and IMSIM [4].

8.2.2.3 CSS II: This simulator was developed by IBM to support its own system analysis needs, and to aid in analysis of customer facility requirements.

IBM now provides CSS II as proprietary software on a rental basis. CSS II is similar in concept to GPSS but differs in one important aspect: it is not general but applies specifically to computer systems. Thus its language speaks in terms of tape units, disk files, communication lines, and terminals, and provides instructions for the modeling of programming systems.

CSS programming consists of a specification of system elements, a specification that generates job streams, and specification of the logical operations to be performed on the job elements. Its generality is enhanced by permitting a more or less complete construction of both the system hardware configuration and the software operating system, to a level dependent on the user's needs and interests.

8.2.2.4 IMSIM: IMSIM was developed by Systems Development Corporation for the NASA Manned Spacecraft Center. It presents a less general approach to computer simulation, in comparison to CSS, because user constructions are confined to the preparation of input tables which define the configuration of computer system elements and the job stream. The algorithms that define the

software operating system cannot be modified, except for a few switch setting choices. The operating system programmed into IMSIM includes the capability of simulating priority-dependent multiprogrammed and multiprocessor computing systems. IMSIM is supported only at the Manned Spacecraft Center, NASA. It is written in Modlit, a language similar in many respects to GPSS.

8.3 Phase 2 -- Low-Level, Detailed, Mixed Simulation

The attractiveness of high-level simulation lies in its ability to discover major conceptual flaws before the design is committed to hardware and before the operating system software is frozen. Hopefully, this effort also builds confidence in the system concepts at a low cost. The major shortcoming of high-level simulation is that design flaws may have been obscured due to simplifications in the models employed.

The low-level simulation employing various degrees of real hardware, software and a simulated environment will provide a more definitive verification of system performance, albeit at a significantly higher cost. The CVT program presently being carried out at MSFC is an example of a simulation with a real computer and data bus. The space station environment and typical work loads will, however, have to be simulated by artificial means.

8.3.1 The Simulation Process

The simulator is a device (both hardware and software) which provides the developer of the system with overall external control of the system being tested. The simulator provides hardware and software required for specifying, monitoring, and testing the system under well controlled conditions. Reference 1 describes the simulation process which can be organized into four factors as shown in Figure 8.1. These are:

a) the user (USER),

b) the simulator itself (SIMULATOR)

c) the computer system being simulated (SYSTEM), and

d) the simulation output (OUTPUT).

Let it be made clear that the SYSTEM being simulated may be implemented as either a complete software effort on a host computer or it may contain certain elements of real hardware

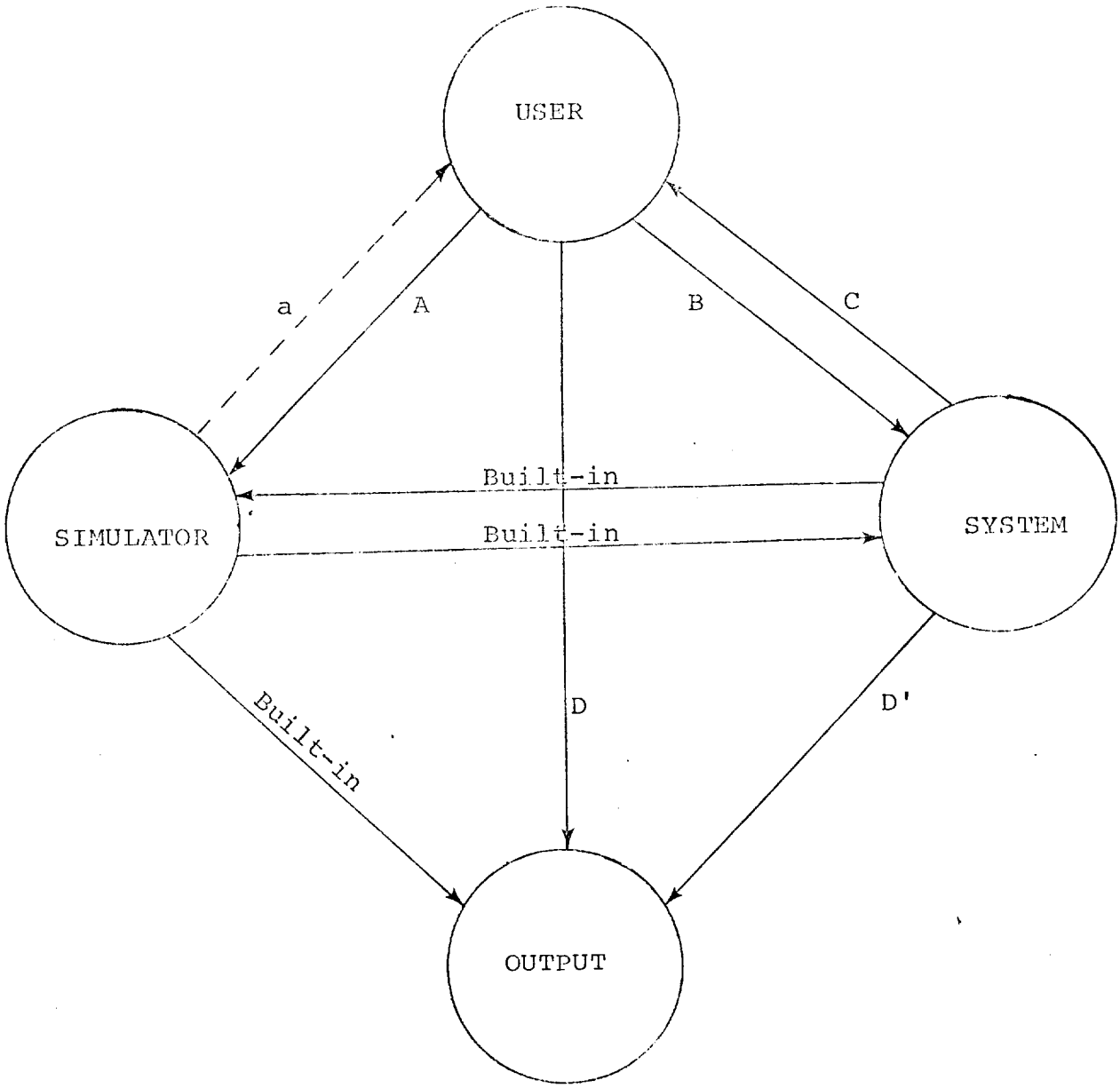


Figure 8.1: Simulator Logical Partitions

and software. There are advantages and disadvantages to both approaches. These shall be made clear in the discussions to follow.

The geometry of the logical partitions in the simulator is shown in Figure 8.2, and the physical control is shown in Figure 8.3 following. The control path labeled A in the two figures provides the user with the capability of specifying the load module to be simulated, start-location and initial SIMULATOR clock setting, the maximum allowable SIMULATOR clock setting (to assure run termination), the configuration of the SYSTEM (levels of redundancy, numbers of spares, initial fault states, etc.) information relative to automatic reconfiguration, illegal instruction detection, execution of instructions in read/write memory, etc.

The primary control, which the USER specifies, follows path B. By this path, and the return path C, the USER will be capable of ordering entry to routines which he provides, upon the occurrence of events or situations he specifies. The trigger-directives can include time conditions, location reference (instruction or operand access), and state changes (I/O, interrupt, hardware error detection signals, etc.). Once his routines have been entered as a consequence of a trigger directive, the USER is capable of accessing all locations, registers, states, and conditions in the SYSTEM, and modifying them as he sees fit. Through an interface language, the USER may implement actions based upon conditions of almost arbitrary complexity, by simply programming the testing of these conditions in his routines.

Control paths D and D' provide information for OUTPUT, such as trace, flow-trace (output produced by branches only), interrupt-occurrences, faults, or output directly from the USER.

Information is not required on path "a" since the USER only interacts with the SYSTEM once the run starts and needs no interaction with the SIMULATOR. Figure 8.3 shows that the SYSTEM is actually implemented within the SIMULATOR, and that the control paths to it actually interact via the SIMULATOR.

Path E of Figure 8.3 represents the closed-loop dynamic flow capability which the USER can exercise within his interface-language routines. These routines may, in turn, call routines prepared in other languages to perform further processing. Using external routines via this path allows the convenient addition of a data-recording capability to the system to allow post-run processing and the addition of almost any conceivable environmental model.

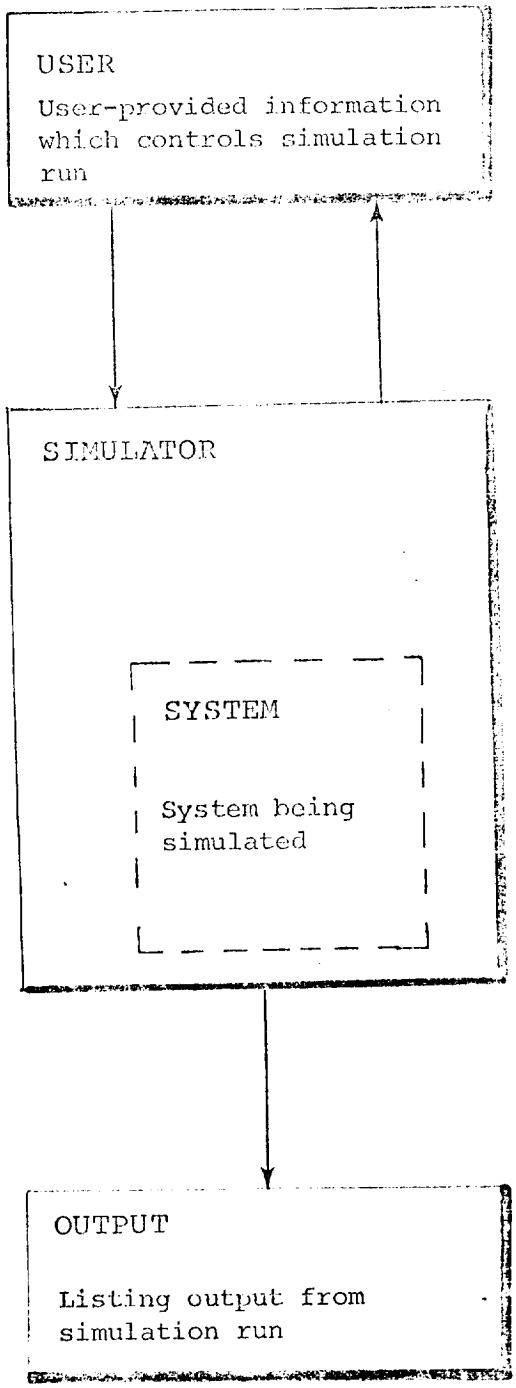


Figure 8.2: Basic Simulator: Input, Simulator, Output

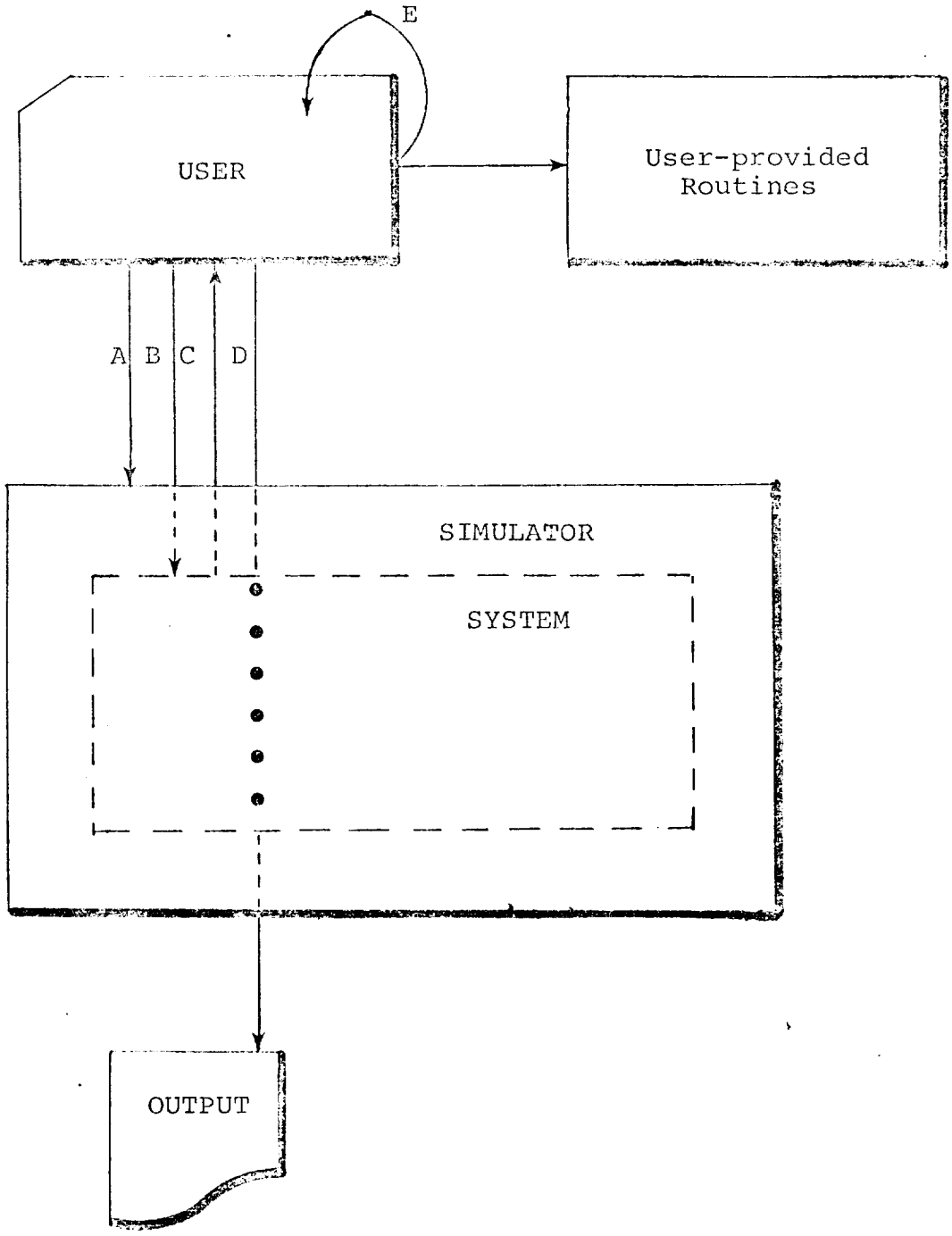


Figure 8.3: Simulator Physical Control Flow

8.3.2 Simulator Design Issues

8.3.2.1 User Interface: For any simulation effort to be successful, the user or experimenter must be provided with a capability of exercising complete control over the simulation from beginning to end. This control includes the ability to:

a) Specify all initial conditions including default conditions, before the simulation is run. This includes the ability to specify the contents of memory locations, control bit and processor registers.

b) Specify the work load to be run in the system, including hardware elements to be used.

c) Specify the environment to be simulated, including extra-ordinary events such as failures.

d) Specify the outputs to be generated and reported.

e) Specify modes of operation, the ability to roll back, and snapshot times.

8.3.2.2 Work Load: The simulation of the processing unit or employment of real hardware is only the first step in the simulation of a computer system. In order to provide meaningful information on complex system interaction a "work load" for the system must be specified. For the SUMC MP this will include a reasonably complete set of actual or simulated applications software modules as well as the real executive system.

If one attempts to evade the issue of generating a realistic work load, many important design factors may be overlooked. For example, if a simulated work load is generated by a collection of subroutines, each one occupying a given amount of memory space, and a specified execution time (as simulated by a countdown loop), the information concerning instruction frequency is lost. Also, since memory requirements for each subroutine are assigned arbitrary values, many factors concerning memory management become distorted.

It is suggested that an effort be made to generate the real application software to be used as the work load. Space qualified software is not required for a system simulation. Therefore, the use of real applications software, to the extent permitted by the simulator's limitations, may be less difficult than trying to generate a realistic model of the work load.

Because of the interaction between hardware and the executive, it seems only reasonable that the executive system model must contain as many as possible of the features of the real executive. A large number of the parameters or algorithms which will be modified because of the simulation experience are implemented in the executive software.

8.3.2.3 The Environment: In simulating aerospace computer systems, the work load must often interact with the spacecraft and its environment. For example, navigation programs must receive accelerometer inputs before they can correctly update vehicle position and velocity. A high degree of similarity must be maintained between the real and modeled environments so that the simulated computer can be subjected to computational loads and dynamic situations closely approximating the conditions of the actual mission.

The simulation environment developed for the Apollo Guidance, Navigation and Control System included modeling spacecraft dynamics, engines, optics, astronaut interactions, atmospheric and gravity effects, motions of celestial bodies, etc.

For the SUMC MP, the environment cannot be simulated within the SUMC itself. This would distort memory management, I/O, processor allocation and real time factors. The simulated environment must be provided by external equipment. For example, the H316 computer can provide such a vehicle by simulating the data bus and all its peripherals. If a real data bus is employed with a limited amount of real avionics equipment then the H316 could be interfaced to the data bus to simulate those equipments which are impossible to exercize satisfactorily in the laboratory (e.g., IMU's, fault detectors within BITE).

8.3.2.4 Measurement of the System Under Test: The accumulation of statistics and the output presentation of this data are the ultimate result of any simulation result. If a real computer is used instead of a simulated model then a major problem can arise due to the lack of computer memory capacity for trace and dump routines and data. If the memory is used for trace and dump data then it cannot be used to process the workload. The results of the simulation run will therefore be distorted.

A secondary problem also arises in that real time aerospace computers usually do not possess a full complement of high speed record recording equipment, such as card readers, high speed printers, or tape units. The attachment of these equipments could also distort the results since they put an abnormal load on the I/O.

A complete software simulated system will not suffer the problems mentioned above since time and memory space are also simulated entities. When real hardware is used within the simulator, it is difficult to compensate for inadequate memory or the loss of real time.

Three features which were incorporated into the Apollo computer simulator are presented below as examples of the interaction of the simulator and the simulated system. These interactions imply that if a real SUMC MP is to be employed as an element of the simulated system, a design effort must be undertaken to provide the correct "hooks" into the hardware so that useful results may be obtained.

a) Rollback

A useful feature to be used in microsimulation is rollback [5]. Long missions such as Apollo require simulation time on the order of hours. Should the host computer (on which the simulation is being executed) malfunction, the simulation will abnormally terminate. Upon restart one does not want to go back and duplicate the execution of this simulation from the beginning of flight. By establishing rollback points in the simulation this problem is avoided. At rollback times complete core and register dumps are taken, and this information is put on a secondary storage device. Then upon system failure the simulation can be restarted at the last rollback point by loading memory with this stored information. The overhead associated with rollback is well justified with long simulations, such as Apollo. However, to prevent this overhead from becoming too high the system designer must decide upon a judicious criterion for establishing rollback points. That is, he must trade off the cost of frequently storing rollback information with the savings in not having to resimulate a large part of the flight.

b) Stress Testing

Stress testing can be provided in a simulator to help determine if combinations of application programs will exceed their combined time budgets under the executed conditions of operation. This request reduces the speed of the object computer. If a group of application programs is run in a simulation with a computer whose speed is, say, 75% of the real computer capability, successful operation may be interpreted to mean that no more than three-fourths of the computer capacity has been absorbed. This special request can thus

be used to "diminish" the capability of the computer until a point is reached where timing requirements are not satisfied. This level then is a guide to the amount of computer capability still available for other software.

Stress testing can also be used to verify the "thrashing" threshold of memory management. If the amount of available memory is reduced but the workload and environment are held constant then a measure can be obtained as to how much excess memory is available for multi-programming.

c) The Coroner Request

A "coroner" special request can be implemented in a simulator for post-mortem diagnosis. The request causes storage of information from each simulated instruction in a circular buffer of size n. If the run abnormally terminates, a list of the last n instructions simulated is produced. This list is a valuable aid in determining the reason for the abnormal termination. However, the overhead associated with this request requires that it only be used when its cost is outweighed by the enhancement of debugging efficiency.

d) Knobs and Dials

A system simulation is undertaken not only to verify specific design concepts, but also to make performance measurements under various parametric conditions. In order to achieve this objective the system (hardware and software) must be provided with enough flexibility (knobs and dials) so that the various design parameters may be adjusted.

Although the details of the SUMC MP have not been published by MSFC a number of suggestions can be made concerning those entities which should remain as variable parameters during the simulation. Implicit in the following listing are obviously a number of assumptions which, if incorrect, could make the variable unnecessary. For example, if a management directive exists that only two processing units are to be employed with no concern for future expansion then a number of problem areas associated with multiprocessor design degenerate into trivial solutions.

The following list describes some of the design parameters which should be kept variable during the low level simulation process.

1) Operating Memory (M2)

Assume a paged virtual memory concept is employed. The following items should be adaptable in order to optimize performance.

i) Page size

ii) Page replacement algorithm

iii) Page presence algorithm. If an associative memory is employed to determine the presence of a page in M2, then the number of words in the associative memory should be made a parameter.

iv) Total size of M2 storage as well as the number of M2 modules.

v) Possibly the speed ratio between M2 and M3.

2) Processing Unit and Local Storage (M1)

i) Instruction architecture. A measure of instruction frequency will indicate which instructions are not needed. Similarly the measurement of subroutine usage of various control features will indicate which instructions need to be incorporated into the design.

ii) Depending upon the use of M1 its size should be variable.

iii) The algorithm used to assign processes to processors should remain a variable as should most of the executive functions dealing with resource allocation.

3) Communication

i) The P-M2 internal bus width and rate should be changeable especially if a bottleneck is anticipated, based upon phase 1 simulation.

ii) The communications link from processor to processor as well as from processor to I/O should be made flexible so that the traffic capacity can be increased if a bottleneck is discovered.

References for Chapter 8

1) Intermetrics, Inc., Final Report, Contract NAS9-12119, "Advanced Data Management System Analysis Techniques Study", July 1972.

2) Gordon, Geoffrey, System Simulation, (Prentice-Hall, Englewood Cliffs, New Jersey, 1969).

3) IBM, CSS II General Information, Technical Publications Department, 1133 Westchester Avenue, White Plains, New York.

4) System Development Corporation, "Information Management System Design For Future Missions, Users Manual", (Report TM-(L)-4719/001/01, Contract NAS9-11211, NASA Manned Spacecraft Center, Houston, Texas).

5) Chandy, K.M., and Ramamoorthy, C.V., "Rollback and Recovery Strategies for Computer Programs", (IEEE Trans. on Comp., C-21(6), June 1972), pp. 546-555.

Chapter 9

CRITIQUE OF SUMC's ARCHITECTURAL CHARACTERISTICS

9.1 Design Goals

A critical evaluation of the SUMC design is provided in order that future efforts may have the benefit of the present study results. This critique will not be primarily directed at the implementation aspects of the circuit and/or logic design, but rather at the higher level architectural features of the hardware. An evaluation of any design must of necessity rest with a determination of how well the design approaches a set of goals. Therefore, a set of design goals is now presented which is Intermetrics' interpretation of MSFC's desires in the development of the SUMC project.

a) The MSFC desire to use a basic SUMC hardware design on a wide variety of missions, which will require a wide range of computation power, leads to the requirement for a hardware design which is expandable. "Expandability" should be considered with respect to such features as word length and sizes of the various memory and processing structures, including the micro memory, scratch pad, ALU, multiplexers and main memory.

b) The variety of application requirements leads to a desire to create an architecture which is flexible and adaptable to changing conditions. For example, the instruction set should be able to be modified or changed. Similarly, components should be able to be utilized within the same architectural structure regardless of their execution speed. As various technologies improve, this then allows the smaller and faster logic and/or memory elements to be incorporated into the design with a minimal impact.

c) A specific requirement of the SUMC expandability and adaptability design is the ability to utilize the design as either a stand-alone uniprocessor or as a larger multiprocessor system.

d) The "U" in SUMC stands for "ultra" reliability. This must not only include the ability to operate for a long period of time without failure, but also (from a

practical point of view with respect to current technology), must indicate the ability to detect failures. The detection of failures is required if a multiprocessor is to constrain error propagation and possess the ability to reconfigure.

e) Since the SUMC family of computers is meant primarily for aerospace applications, the conservation of weight and power becomes of primary importance.

Keeping in mind these different critieria, the following sections will examine various aspects of the SUMC design. Not all of the aspects are independent of each other, but they are presented in such a manner so as to highlight different points of view.

9.2 Micro Instruction Sequencing

In a microprogrammed machine where flexibility is one of the objectives, it is extremely important that the micro sequence control itself be flexible. The sequencing control presently available in SUMC is described in Figure 9.1. The only control actions possible are

a) stepping thru the micro code (0., 1., 2., 3.)

b) branching to a location described by

   1) an ALU output (4.)

   2) associated with an opcode (5.)

   3) given in the micro code (13.)

c) alternate choice in either

   1) branching or holding (6., 7.) or,

   2) branching or stepping (8., 9., 10., 11., 12., 14., 15.)

Although these forms of sequencing do allow the generation of a static set of linked micro code, they do not allow for easy modularization of micro code.

While this feature becomes particularly important when the instruction architecture contains powerful semantically concise operations, it is also extremely important with standard current forms of instructions. The execution of an

CONDITIONS	SEQ. ACTION	I.C. ACTION	Binary Code
None	+1	Hold	0000
None	+1	IR (26-31)→IC	0001
None	+1	MROM (C11-C16)→IC	0010
None	+1	PRM (26-31)→IC	0011
None	PRM (22 31)→SEQ	Hold	0100
None	IAROM SEQ	Hold	0101
IC>0	Hold	IC - 1→IC	0110
IC=0	MROM (C7-C16)	Hold	
IC≥4	Hold	-4	0111
IC<4	MROM (C7 - C16)	Hold	
CNT = 1	MROM (C7 - C16)	Hold	1000
CNT = 0	+1	Hold	
INT + DOT = 1	MROM (C7-C16)	Hold	1001
INT · DOT = 0	+1	Hold	
INT Req. = 1	MROM (C7-C16)	Hold	1010
INT Req. = 0	+1	Hold	
INT+DOT+DIN = 1	MROM (C7-C16)	Hold	1011
INT·DOT·DIN = 0	+1	Hold	

CNT = EALU overflow or ALU overflow or DEX3 as specified by ACCS, CNT field

ACCS = 1	+1	Hold	1100
ACCS = 0	MROOM (C7-C16)	Hold	

ACCS = PRM sign or ER sign as specified by the ACCS, CNT field

None	MROM (C7-C16)	Hold	1101
IC>0	+1	-1	1110
IC=0	MROM (C7-C16)	Hold	
IC≥4	+1	-4	1111
IC<4	MROM (C7-C16)	Hold	

Figure 9.1: Control Conditions and Actions

instruction can be viewed as occuring in three phases: instruction fetch, operator decode, and execution of the operation.

The SUMC allows for common manipulation of all instructions in both the instruction fetch and operator decode phases of execution. It is interesting to note that after the instruction has been fetched, the memory operand is fetched, if the operator is of the appropriate "class" of instructions. This differentiation is performed by the hardware and is completely dependent, therefore, upon both the instruction architecture and its physical bit mapping. There is no general way to have several classes of instructions, each with its own idiosyncrasies, without this special hardware help. This is because the decision on whether or not to read memory must be performed in the "common" section of code.

If there were to be the ability to call and link in the micro code, then the question as to whether to read an operand from memory could be decided after the operator had been decoded and the execution of the operation had been entered.

(The one current possibility for modularization within the SUMC micro code would be:

a) Place the return micro address in the PRM

b) Branch to the micro sub-routine

c) Upon entering the subroutines, save the return address in the SPM

d) To return, gate the return address from SPM to the PRM and into the SEQ.

This would effectively take four micro words.)

Besides the desire for micro code modularization for complex instruction sets, the next section will point out the need to be able to do much more micro condition testing for sequence control.

9.3 Choosing Functions to Optimize

It has been observed that the SUMC hardware has been optimized for the implementation of the multiply, divide and square-root operations. However, what is the actual expected percentage of occurence of these operations? In particular, what is the frequency of distribution of all the implemented machine instructions?

C.C. Foster et. al. [1] has made a study of OP code usages on the CDC 3600 and has found that in scientific Fortran Programs, the compiled code contained only 10% arithmetic instructions. The remaining instructions were involved with load, store, subroutine linking and various other control operations. For the more commercial type of application, the total percentage of all arithmetic instructions fell to less than 5% [2]. The most common arithmetic operation was clearly addition. Even for the program with the most arithmetic functions, multiply and divide were less than 2%.

C.C. Church [3] states:

> "In instruction occurence we found arithmetic 8.3 percent and jumps 12 percent. What are we doing with the rest of the commands? Obviously, we need the "Data Move" function, but do flow charts call for anything near 40 percent? And what of the transfers: My flow charts do not call for anything near 23 percent of the problem to be involved in transferring."

While these types of statistics can be interpreted as indicating a mismatch between the problem to be solved (i.e., the program) and the operations provided (i.e., the machine instructions), they can also provide insight into the design and implementation of instruction sets. If the instruction set provided is of the current machine level form (e.g., IBM 360) then, for example, the multiply and divide instructions are not driving design features. If these instructions are truly less than 2% in occurrence, then their optimization and reduction of their execution time by half will only save 1% of the overall execution time. On the other hand, an optimization of branches by half their execution time would make a dramatic savings in actual execution time.

While it is understood that certain data reduction or filtering problems do require an above normal amount of multiplication, this is not a common occurence and the multiply and divide instructions should not form the basis of the machine architecture.

If one takes the SUMC JZ (Jump Zero) instruction for an example, (Figure 9.2) it can be seen that not only can it be made faster, but the number of micro instructions can be reduced if necessary conditions to be tested are generated by the hardware. The testing of conditions is indeed the method of determining control flow through an algorithm, and therefore, will always either have to be in some fashion artificially produced or explicitly tested. The cost of providing these extra dynamic conditions is small when compared to the gains in execution time and savings in micro-memory.

If (A)=0, GO TO Z

If (A)≠0, GO TO NI

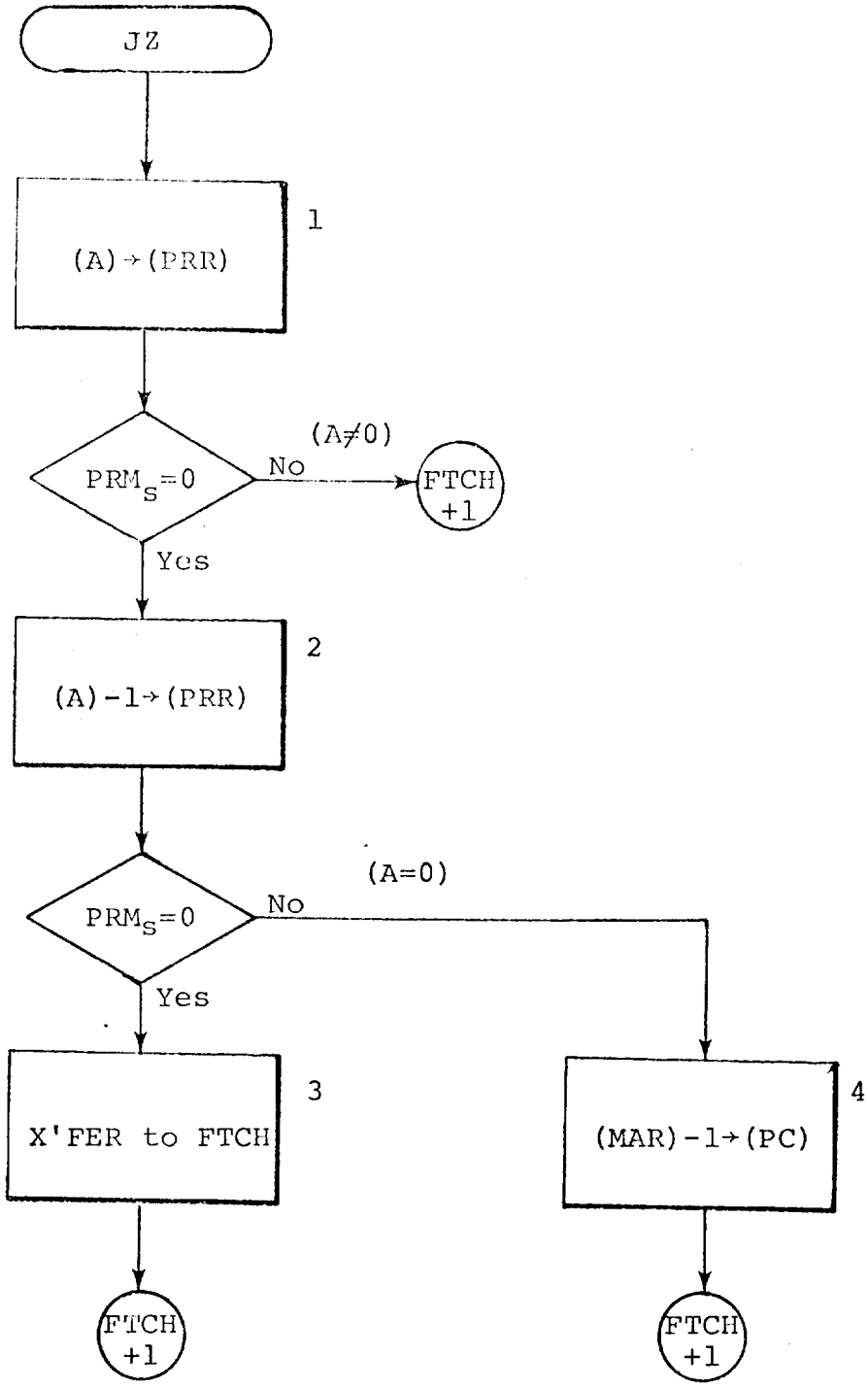


Figure 9.2: Micro Program Flowchart
(Jump Zero)

It can be noted that further savings can be had in the revised JZ flow chart (Figure 9.3) by either placing the (MAR)-1→(PC) function as a special entrance to the FETCH routine, since it occurs in several SUMC instructions, or for this same reason have this action as part of a sequence control state.

9.4 Field Manipulation - Maskings - Shifting - Bit Addressing and Shifting

The word length of a computer is often chosen because of arithmetic precision and, cotemporaneously, the instruction format size. Once chosen, this word length then becomes an artificial quantum of addressability. This is the case with the SUMC. The implementation of an instruction set often requires the efficient manipulation of variable length fields, masking, bit manipulation and testing. While the SUMC can accomplish all these functions at a Macro level by using shift and logical instructions, it is suggested that if flexibility is to be obtained the high frequency of use of these functions in various instruction architectures requires that they should be more directly under Micro level control.

The SUMC does recognize this fact, in a limited way, by providing in the hardware the extraction of mantissa, characteristic and sign of floating point arithmetic words stored in scratch pad. However, this is rigid. The hardware design should not initially presume to know the desired arithmetic precision of the application. For example, the queues and control bits required for the executive functions of the SUMC are not given special hardware since they are not known in advance.

What is desired is a generalized bit manipulation, masking, field insertion and extracting mechanism which can be micro controlled. In the actual implementation of a particular instruction set for a particular mission, it is recognized that this generally could be specialized in order to optimize the actual usage. An example of the need for testing of certain bits efficiently would be if it were decided to implement indirect addressing and hence the "indirect" bit of the operand would have to be efficiently known during the effective memory address calculation. Besides changes in the meaning of instruction fields, it could also be possible to realize other data types or other physical forms of current data types.

9.5 Limited Scratch Pad Addressing

The philosophy of a generalized register set contained in a scratch pad structure is very good as far as providing an

If (A)=0, GO TO Z

If (A)≠0, TO GO NI

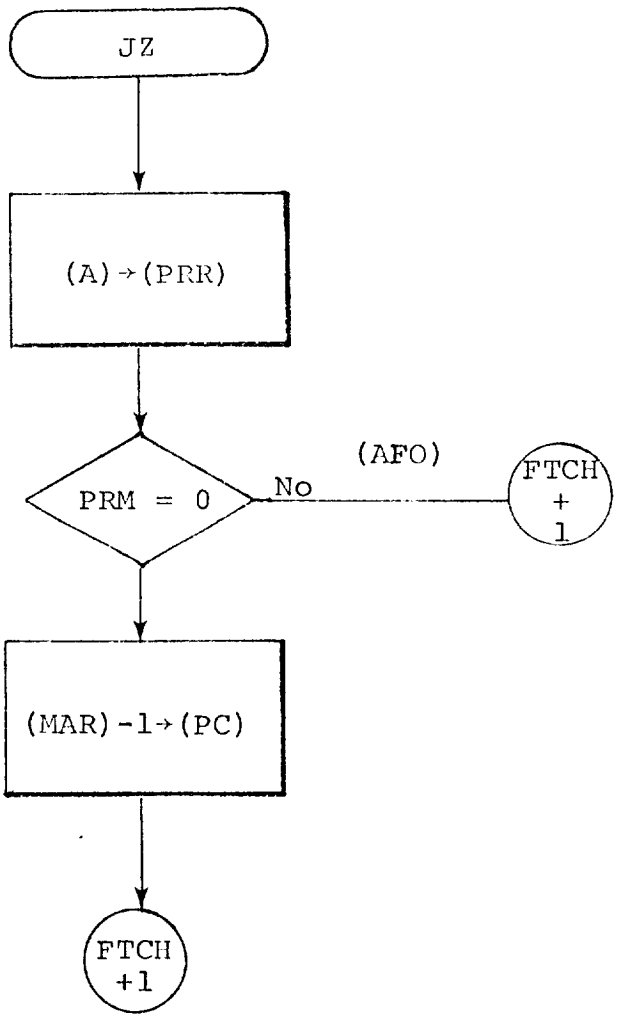


Figure 9.3: Micro Program Flowchart
(Jump Zero) Revised

adaptable design. It would be desired on the hardware level to allow any register to serve any function. The specification, therefore, as to the assignment of registers would be contained within the micro code. The location within the scratch pad of the macro level program counter, base register, etc., should be specified by the micro program and not dictated by an arbitrary hard wired location. One can easily conceive of instruction sets with automatic base registers or none at all, or with a return address stack. The present SUMC design does not allow this generalization.

The internal interconnection of the scratch pad address register (SPAR) to the Instruction Register (IR) and the micro memory buffer register indicates that addressing of the scratch pad is a completely static operation. That is, it is specified in advance within instruction code or micro memory code. The ability to dynamically deduce or calculate a scratch pad address is not possible because the SPAR can not be loaded from one of the SUMC's internal registers, such as the PRR, MQR or MAR. The dynamic determination of scratch pad address would be required if one wished to implement a stack within the scratch pad.

9.6 Micro and Main Memory Speed Ratio

The current T^2L version of SUMC operates with a micro memory cycle time of 330 nanoseconds, while main memory possesses a 660 nanosecond cycle time. It is suggested that the speed ratio between micro and main memory should be closer to 5 or 10 to 1 instead of 2 to 1. This becomes especially desirable when an instruction set is more complex and semantically powerful than the IBM 360 instruction set. In more powerful instruction sets, one finds both:

a) an instruction operation specified in fewer bits, and hence memory does not have to be read as often, and

b) the operations to be performed are themselves more complex and therefore take more computational steps.

9.7 Main Memory Synchronization

While reviewing the micro code flow charts, it was observed that the processor or micro memory cycle time was synchronized to the main memory cycle time by executing micro level NOPS. The main memory cycle time therefore was an integral part of the micro code. This can be disastrous for two entirely different reasons.

a) If a slower or faster main memory were employed many changes would be required in the actual micro code.

b) In a multiprocessor one can not determine the exact time between a memory request and the response, since the addressed memory module might be busy with another processor and the request might take a number of memory cycles to be satisfied.

Multiprocessors, therefore, can not guarantee their exact response time with respect to memory.

What is required is a completely asynchronous operating memory interface where the execution of micro code and memory timing are not intertwined.

In a multiprocessor environment it is necessary that a process be able to read the contents of a memory location and change its value by writing into it all in one period of time at the exclusion of all other processors. This form of read/write mechanism must be provided by any potential multiprocessor.

9.8 Limited Modularity Concept

The "M" in SUMC, which stands for modularity, seems to extend only to the packaging of arithmetic and register functions into 4 bit entities. The concept of modularity can be extended to the higher level of internal architecture by providing an internal structure which is organized around 1, 2 or 3 buses. These buses allow all the internal structures to communicate between one another. As needed, new structures may be added, such as a floating point unit or an associative memory unit. Most present day mini computers (see Figure 9.4) are designed around an internal bus structure.

This concept can be extended as in the MLP 900 (IC 9000) which also provides what are called program cards. These are hardware modules addressed by micro memory to provide specific hardware functions.

Mini computers such as the HP 2000 series, PDP-11, MODCOMPI, GRI909, etc., are all built around an internal bus structure. Often it is this internal bus structure which enables the system to expand and contract to meet varying requirements.

The "M" in SUMC is severely limited with respect to this described form of modularity.

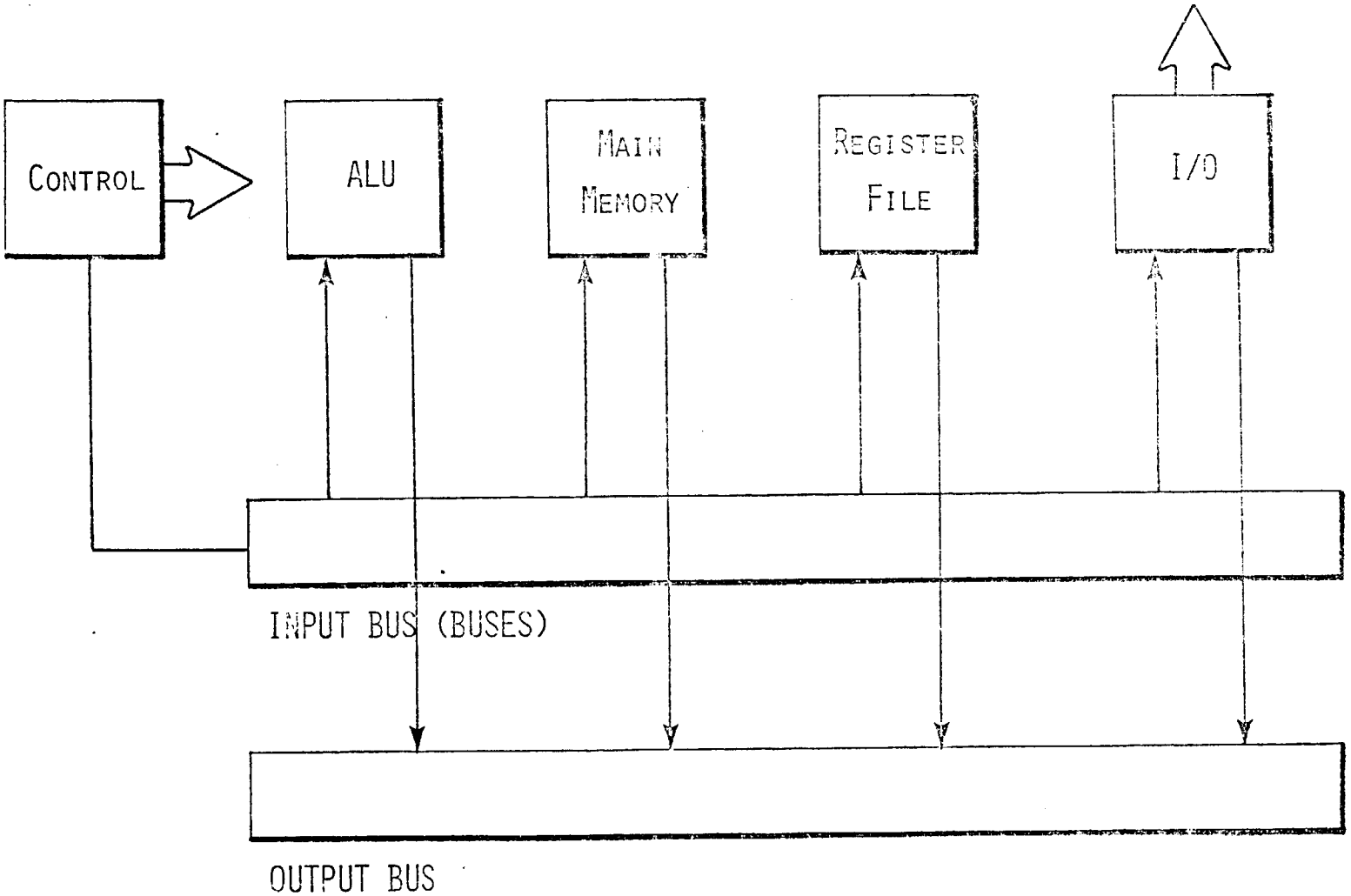


Figure 9.4: Generic Mini Computer

9.9 The "U" in SUMC -- Ultra Reliability

Reliability, clearly requires "good" components. The SUMC program does attempt to achieve component level reliability by experimenting with advanced state of the art component and packaging and fabrication techniques. Reliability is one of the major design goals of the SUMC architecture. This being the case, it is surprising that the architecture of the SUMC does not consider hardware detection of the major fault conditions of integrated circuit implementation. The packaging and definition of the modules should consider the effect of failure and should attempt to make detectable failures more statistically independent. For example, integrated circuit modules should tend to be more bit oriented than function oriented.

It is necessary in reliable systems to have "immediate" fault detection within the hardware in order to prevent propagation of errors. The interaction of transient faults and the micro execution of instructions must be carefully considered, and made part of the basic structure.

9.10 Confusion Between Design Levels

A basic philosophical comment seems appropriate. A truly modular design should possess maximum independence between design levels. That is, the architecture (block diagram) level, instruction definition level, and the implementation (logic design, circuit technology) level should be approached as independently as possible. A change of definition at one level should not cause major impact on the other levels.

When flexibility is desired the implementation architecture should be generalized enough to allow the implementation of a wide variety of instruction sets. This is particularly true when one considers a large future time framework. While most current instruction architectures are similar to the 360, they will become more and more problem oriented such as the Burroughs B6700. The instruction set should reflect the major application to which the system is to be used. For example, when a Higher Order Language is employed, the instruction set should be so specified as to aid in the generation of, and hence the efficient execution of, compiled code.

Similarly, the introduction of new technology at the implementation level of design affects speed, weight, power and cost, but should have no major impact upon the instruction set or (processor, memory, I/O) architecture.

Clearly one can not be too pedantic in the utilization of the principle stated above and must appreciate the practicalities of all design levels.

The SUMC design has greatly intertwined the (processor, memory, I/O) architectural, instruction set definition, and implementational levels of the design.

References for Chapter 9

1) Foster, C.C., et. al., "Measure of OP Code Utilization", IEEE Trans. on Comp., May, 1971, pp. 582-584.

2) Bingham and Kauffman, "Analysis of Static Object Code Produced by Algol and Cobol Compilers for the Burroughs B5500", Burroughs Corporation, Paoli, Penna., February, 1969.

3) Church, C.C., "Computer Instruction Repertoire-Time for a Change", SJCC, 1970.